# APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities.

Sébastien Salva, Stassia R. Zamiharisoa

**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities. ⋆

**Sébastien Salva**[1], **Stassia R. Zafimiharisoa**[2]

[1] LIMOS - UMR CNRS 6158, University of Auvergne, France, e-mail: sebastien.salva@udamail.fr
[2] LIMOS - UMR CNRS 6158, Blaise Pascal University, France, e-mail: s.zafimiharisoa@openium.fr

Received: date / Revised version: date

**Abstract.** The Android messaging system, called intent, is a mechanism that ties components together to build applications for smartphones. Intents are kinds of messages composed of actions and data, sent by a component to another component to perform several operations, e.g., launching a user interface. The intent mechanism offer a lot of flexibility for developing Android applications, but it might also be used as an entry point for security attacks. The latter can be easily sent with intents to components, that can indirectly forward attacks to other components and so on. In this context, this paper proposes APSET, a tool for Android aPplication SEcurity Testing, which aims at detecting intent-based vulnerabilities. It takes as inputs Android applications and intent-based vulnerabilities formally expressed with models called vulnerability patterns. Then, and this is the originality of our approach, class diagrams and partial specifications are automatically generated from applications with algorithms reflecting some knowledge of the Android documentation. These partial specifications avoid false positives and refine the test result with special verdicts notifying that a component is not compliant to its specification. Furthermore, we propose a test case execution framework which supports the receipt of any exception, the detection of application crashes, and provides a final XML test report detailing the test case verdicts. The vulnerability detection effectiveness of APSET is evaluated with experimentations on randomly chosen Android applications of the Android Market.

**Key words:** security testing – model-based testing – Android applications – intent mechanism

## 1 Introduction

Security vulnerabilities are common issues in any complex software system. Several recent security reports [2] and research papers [7,11] show that mobile device operating systems and applications are no exceptions. Smartphones can be customised by owners and enriched with a large set of applications available from stores. These features undeniably augment the exposure to attacks. This is why more and more users also expect their personal data to be protected and their applications to be isolated from malicious ones. The Android platform, which is often cited by the previous reports, introduces some mechanisms to secure applications: it supports sandbox isolation by launching applications in their own security sandboxes and an application-level permissions model. However, Android also opened the sandbox by adding the possibility for applications to communicate with one another. These applications consist of components that are joined together by means of a composition mechanism called *intent*, which corresponds to an inter-application and intra-application communication mechanism used to send messages, to call or launch another component. This mechanism, combined with permissions over components, offers a lot of flexibility to develop applications. Nevertheless, some papers showed that permissions can be bypassed. In this context, some tools have already been proposed to check the permission validity [11]. Even with the right permissions, applications can be still vulnerable to malicious intents if incautiously designed [7]. Indeed, the contents of intents can also be sniffed, or replaced by malicious applications to inject incorrect data or attacks. Such intents become the key to crash other applications, to inject or extract personal data, to call paid services without user consent, etc.

Detecting and reducing the risk for Android applications of being vulnerable to malicious intents can be done by testing. Whatever the testing technique employed, e.g

⋆ This work was undertaken in collaboration with the Openium company (http://www.openium.fr). Thanks for their valuable feedbacks and advices on this paper.

Unit testing or Model-based testing, the detection of security flaws requires a significant allocation of time and resources to code test cases or to write formal specifications from which test cases can be automatically derived. The use of effective tools can strongly decrease the testing cost though. Clearly, only few tools are currently available and these have limitations. In this context, this paper presents APSET, a tool for Android aPplication SEcurity Testing, which aims at detecting intent-based vulnerabilities. APSET takes vulnerability scenarios, formally expressed with models called *vulnerability patterns*. The latter are specialised ioSTS (input output Symbolic Transition Systems [12]) which help define test verdicts without ambiguity. Given a set of vulnerability patterns, APSET performs both the automatic test case generation and execution. The test case construction is achieved by automatically generating class diagrams and partial specifications from the information provided in the Android documentation, the component compiled classes and the configuration files found in an Android project. This class diagram and specification generation, which does not require human interaction, is an innovative contribution of this paper. It helps automatically determine the nature of each component and describe the functional behaviour that should be observed from a component after receiving intents. These partial specifications also refine the test result with special verdicts notifying that a component is not compliant to its specification. Furthermore, we propose a specialised framework to execute test cases in isolation on real smartphones or emulators. It supports the receipt of any exception, the detection of application crashes, and provides a final XML test report detailing the test case verdicts.

The paper is organised with four sections. The next section presents the tool and gives some insight into its design and architecture. We introduce the test case generation steps, the use of a Model-based technique and the interest of generating partial specifications from Android applications to generate test cases. We also present the framework implemented to execute test cases and to produce test reports. These different steps and the theoretical background of the testing method are then detailed in the remainder of the paper. In Section 3, we recall some definitions and notations related to the ioSTS model. From these, we detail how vulnerability patterns can be written with notations derived from the Android documentation. In Section 4, we continue with the test case generation from vulnerability patterns and partial specifications. We show how partial specifications and test cases are constructed. We also define the test verdict. Afterwards, we show some experimentation results in Section 5 on Android applications developed by the Openium company and on other popular applications such as Google Maps. We show that our tool generates and executes hundreds of test cases that detect security flaws in a reasonable time delay. Finally, Section 6 compares our approach with some related work and we conclude in Section 7.

## 2 Presentation of APSET

### *2.1 Android application overview*

Android applications are written in the Java programming language and are packaged under the form of an .apk file (possibly encrypted), composed of compiled classes and configuration files. Particularly, the *Manifest* file declares all application requirements, the components participating in the application and the kinds of intents accepted by them. Android applications are built over a set of reusable components, having a different role in the overall application behaviour and belonging to one of the four basic types:

- *Activities* are the most common components that display user interfaces used to interact with an application. An Activity is started with intents, displays a screen and may eventually return a response to the calling component,
- *Services* represent long tasks executed in the background. Services do not provide user interfaces to interact with. They are started with intents by other components (usually Activities) and are then bind to them to perform several interactions,
- *ContentProviders* are dedicated to the management of data stored in smartphones by means of files or SQLite databases. Although data could be directly stored in raw files or databases, the ContentProvider component represents a more elegant and secure solution which makes data available with the proper permissions to applications through an interface. ContentProviders are not launched by intents. Furthermore, without permission (the default mode), data cannot be directly accessed by external applications,
- *BroadcastReceivers* are components triggered by intents broadcasted in the Android system which run a short-lived task accordingly in the background. They often relay intents to other components e.g., Activities or Services.

The inter-component communication among Activities, Services and BroadcastReceivers is performed with intents. An intent is a message gathering information for a target component to carry out an operation. It is composed by a component name, an action to be performed, the URI of the data to be acted on, a category giving a description of the kind of components that should handle the intent, and extras that are additional information. Intents are divided into two groups: explicit intents, which explicitly target a component, and implicit intents (the most generally ones) which let the Android system choose the most appropriate component. Both can be exploited by a malicious application to send attacks to

components since any component may determine the list of available components at runtime. As a consequence, we consider both implicit and explicit intents in this work. The mapping of an implicit intent to a component is expressed with items called *intent filters* stored in Manifest files.

Activities, Services and BroadcastReceivers are directly exposed to malicious intents and may be vulnerable [7]. ContentProviders are not called with intents and sounds more secure. Nevertheless, data can still be exposed by the components which have a full access to ContentProviders, i.e. those composed with ContentProviders inside the same application. These components can be attacked by malicious intents, composed of incorrect data or attacks, that are indirectly forwarded to ContentProviders. As a result, data may be exported or modified. The fact of considering the redirection of attacks among components is also an original contribution of this paper.

At the moment, the intent mechanism is the only one available with Android for calling components on unmodified smartphones. In the paper, we assume that the Android operating system is untouched (non rooted for instance), hence the intent mechanism cannot be bypassed.

### *2.2 The APSET tool*

*APSET* (Android aPplication SEcurity Testing) is a tool, publicly available in a Github repository [1], dedicated to the detection of intent-based vulnerabilities in Android applications. It explores an Android application (uncompressed .apk file) and tests all the Activities, the Services and all the compositions of Activities or Services with ContentProviders found in the application since ContentProviders cannot directly be called by intents but may expose personal data through other components. Test cases are generated from Android applications (compiled components, Manifest files) and vulnerability patterns, which are models describing vulnerabilities.

Indeed, APSET is founded upon a model-based testing method: test cases are derived from vulnerability patterns, written with the ioSTS formalism [12]. These patterns describe vulnerable and non vulnerable behaviours of components by means of symbolic automata, composed of variables and guards over variables which allow to write constraints over the actions that can be performed. Besides, APSET generates partial ioSTS specifications from Android applications with algorithms reflecting the Android documentation. The benefits of using the ioSTS model are manifold: firstly, we reuse some existing ioSTS operators to generate test cases. These operators also make simpler the definition of test relations and test verdicts, which express the detection

[1] https://github.com/statops/apset.git

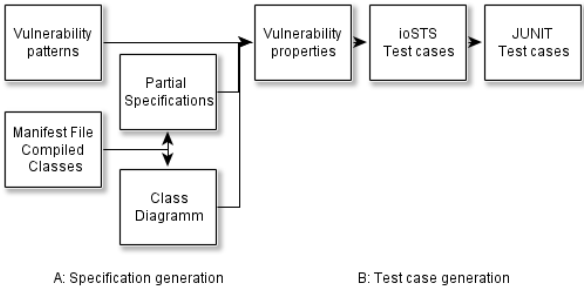


**Fig. 1.** Test case generation

of vulnerabilities in a precise manner. Furthermore, the partial ioSTS specifications help refine final test verdicts by exhibiting if the tested components respect the recommendation given in the Android documentation. They avoid to give false positive verdicts (false alarms) because each component is exclusively experimented with the test cases generated from its specification. Consequently, test cases do not reject a non vulnerable and compliant component. Last but not least, after discussion with several Android developers and testers of the Openium company, we concluded that the ioSTS model is flexible enough to describe a large set of intent-based vulnerabilities and is still enough user-friendly to express vulnerabilities that do not require obligation, permission, and related concepts.

Once test cases are executed, APSET yields the verdict *VUL* when it detects that a component is vulnerable to attacks described in vulnerability patterns, *NVUL* if no vulnerability has been detected or *Inconclusive* when a vulnerability scenario cannot be tested. *VUL/FAIL* and *NVUL/FAIL* are other specialised verdicts, which still indicate whether a vulnerability has been detected or not. These verdicts complete *VUL* and *NVUL* and are assigned when a component under test does not respect its partial specification (hence the recommendations provided in the Android documentation). For instance, an Activity, called by intents composed of the PICK action, has to return a response. If no response is returned while testing, this Activity does not meet the Android documentation. A verdict composed by FAIL is then given.

The tool architecture is based upon two main parts, the test case generator and the test case execution framework:

- Figure 1 depicts the main steps of the test case generation. APSET takes a set of vulnerability patterns by means of the interface depicted in Figure 2. These vulnerability patterns are provided by an Android expert or developer and stored in DOT files. This expert can define manually vulnerabilities from those exposed, for instance, by the OWASP organisation [25]. If the source code of the application is known, precise vulnerability patterns composed of specific values can be also given as inputs. We chose the DOT format since it is a well-known plain text graph de-

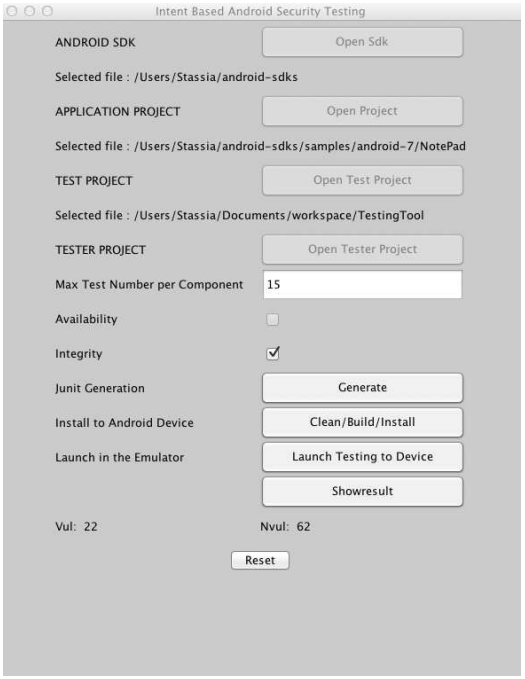


**Fig. 2.** APSET interface

scription language that can be translated into graphics formats. Vulnerability patterns can be then visualised. From an Android application, APSET generates a partial class diagram by means of Java reflection. This one lists the components, gives their types and the associations among them. Furthermore, the partial ioSTS specifications of components, are constructed from the class diagram. These are stored in DOT files and can be visualised as well. Intermediate ioSTS, called vulnerability properties, are then derived from the combination of vulnerability patterns with partial specifications. These properties still express vulnerabilities that are refined with the implicit and explicit intents accepted by a component. Test cases are obtained by concretising vulnerability properties (the latter are completed with values). Finally, the resulting ioSTS test cases, composed of transitions, variables and guards, are translated into executable JUNIT test cases,

– JUNIT test cases are executed with the framework illustrated in Figure 3. It supports test execution on Android devices or emulators. It is composed of the Android testing execution tool provided by Google, enriched with the tool *PolideaInstrumentation* [2] to write XML test reports. The test case execution is launched by an Android Service component which displays test results on the device (or on the emulator). The *Test Runner* starts a component under test (CUT) and iteratively executes JUNIT test cases provided by the previous Service in separate processes. This procedure is required to catch the exceptions raised by the Android system whenever a component crashes.

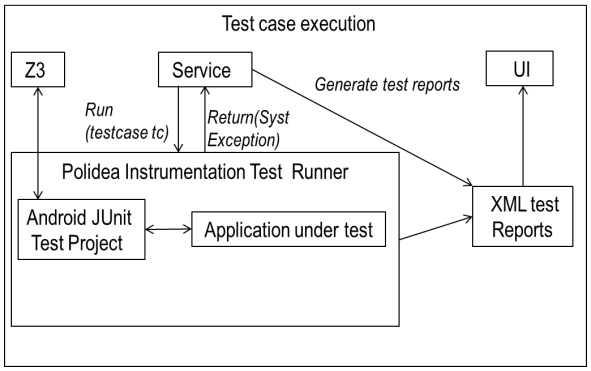


**Fig. 3.** Test case execution framework

Test cases are composed of guards written with the SMT-LIB language. During the test case execution, a solver as to be called to check their satisfiability. The test case execution framework can call two solvers. The first one called *tinysolver* is bundled inside the framework and is used with only straightforward guards composed of conjunctions. Otherwise, the guard solving is performed by the SMT (Satisfiability Modulo Theories) solver Z3 [23], invoked through a REST Web service deployed on an external server. We augmented the SMT-LIB language with new predicates to support String variables and the vulnerability description language presented in Section 3.3. Once all test cases have been executed, the Service displays a screen which summarises the test results and gives the XML test report which details all the assertion results. In particular, the *VUL* and *VUL/FAIL* messages exhibit the detection of vulnerability issues. These reports also offer the possibility of using continuous integration servers like Jenkins [3]. A test result screen obtained on a smartphone is illustrated in Figure 4.

*Example 1.* A part of test report is illustrated in Figure 5. It results from the crash of a component with the receipt of a *NullPointerException* exception. We obtain a *VUL/FAIL* verdict directly deduced from the *VUL/FAIL* message (line 3).

In the next sections, we develop the theoretical background developed in APSET. We describe how to write Android vulnerability patterns with ioSTS. Then, we detail how test cases are derived from vulnerability patterns and partial specifications. A user does not need to be aware of this (hidden) theoretical background though. He only has to write vulnerability patterns.

[2] www.polidea.pl/

[3] http://jenkins-ci.org/

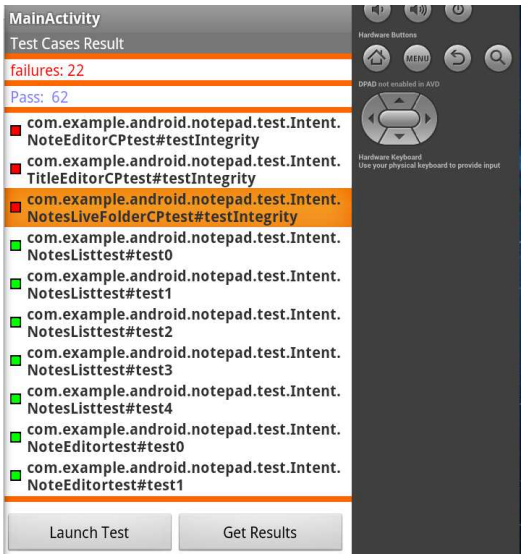


**Fig. 4.** Test results on a smartphone

```
<testsuite errors="0" failures="1" name="
    packagename.test.Intent.ContactActivityTest"
     package="packagename.test.Intent" tests="1"
     time="0.15" timestamp="2013-02-13T10:05:02"
    >
   <testcase classname="packagename.test.Intent.
       ContactActivityTest" name="test1" time="
       0.15">
     <failure> VUL/FAIL
     INSTRUMENTATION_RESULT: shortMsg=java.lang.
         NullPointerException
     INSTRUMENTATION_RESULT: longMsg=java.lang.
         NullPointerException
     INSTRUMENTATION_CODE: 0
     </failure>
   </testcase>
</testsuite>
```

**Fig. 5.** An XML test report

## 3 Vulnerability modelling

Several formalisms have been proposed to represent security policies, properties or vulnerabilities, e.g., regular expressions [26], temporal and deontic logics [9], core typed languages ([6]) or state machines. As stated in the APSET presentation, we shall consider the input/output Symbolic Transition Systems (ioSTS) model [12] to represent vulnerabilities and to generate partial specifications of Android components. This formalism, widely used in the verification and testing areas, can model large and complex systems, e.g., Web services compositions, Desktop or Web applications or critical systems. Therefore, if the Android documentation is enriched or if new vulnerabilities are discovered, the following ioSTS related definitions could be adapted as well. Below, we recall some definitions to be used throughout the paper. Thereafter, we define vulnerability patterns.

### *3.1 Model Definition and notations.*

An ioSTS is a kind of automata model which is extended with two variable sets, internal variables to store data, and parameters to enrich the actions. Transitions carry actions, guards and assignments over variables. The action set is separated with inputs beginning by ? to express actions expected by the system, and with outputs beginning by ! to express actions produced by the system. An ioSTS does not have states but locations.

Below, we give the definition of an extension, called ioSTS suspension, which also expresses quiescence i.e., the authorised deadlocks observed from a location. For an ioSTS $\mathcal{S}$, quiescence is modelled by a new action $!\delta$ and an augmented ioSTS denoted $\mathcal{S}^\delta$, obtained by adding a self-loop labelled by $!\delta$ for each location where no output action may be observed.

**Definition 1 (ioSTS suspension).**
A deterministic ioSTS suspension $\mathcal{S}^\delta$ is a tuple $< L, l0, V, V0, I, \Lambda, \rightarrow >$, where:

- $L$ is the finite set of locations, $l0$ the initial location,
- $V$ is the finite set of internal variables, $I$ is the finite set of parameters. We denote $D_v$ the domain in which a variable $v$ takes values. The internal variables are initialised with the assignment $V0$ on $V$, which is assumed to be unique,
- $\Lambda$ is the finite set of symbolic actions $a(p)$, with $p = (p_1, ..., p_k)$ a finite list of parameters in $I^k (k \in \mathbb{N})$. $p$ is assumed unique. $\Lambda = \Lambda^I \cup \Lambda^O \cup \{!\delta\}$: $\Lambda^I$ represents the set of input actions, $(\Lambda^O)$ the set of output actions,
- $\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by an action $a(p) \in \Lambda$. $G$ is a guard over $(p \cup V \cup T(p \cup V))$ which restricts the firing of the transition. $T$ is a set of functions that return boolean values only (a.k.a. predicates) over $p \cup V$. Internal variables are updated with the assignment function $A$ of the form $(x := A_x)_{x \in V}$ $A_x$ is an expression over $V \cup p \cup T(p \cup V)$,
- for any location $l \in L$ and for all pair of transitions $(l, l_1, a(p), G_1, A_1)$, $(l, l_2, a(p), G_2, A_2)$ labelled by the same action, $G_1 \wedge G_2$ is unsatisfiable.

An ioSTS is also associated to an ioLTS (Input/Output Labelled Transition System) to formulate its semantics. In short, the ioLTS semantics corresponds to a valued automaton without symbolic variable, which is often infinite: the ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and parameter valuations. The semantics of an ioSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow >$ is the ioLTS $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \rightarrow >$ composed of valued states in $Q = L \times D_V$, $q_0 = (l0, V0)$ is the initial one, $\sum$ is the set of valued symbols and $\rightarrow$ is the transition relation. The complete definition of ioLTS semantics can be found in [12]. Intuitively, for an ioSTS

transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an ioLTS transition $(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')$ with $v$ a set of valuations over the internal variable set, if there exists a parameter value set $\theta$ such that the guard $G$ evaluates to true with $v \cup \theta$. Once the transition is executed, the internal variables are assigned with $v'$ derived from the assignment $A(v \cup \theta)$.

Below, we recall the definition of some classic operations on ioSTS. The same operations can be also applied on underlying ioLTS semantics.

An ioSTS can be completed on its output set to express incorrect behaviour that are modelled with new transitions to the sink location Fail, guarded by the negation of the union of guards of the same output action on outgoing transitions:

**Definition 2 (Output completion).**

The output completion of a deterministic ioSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow >$ gives the ioSTS $\mathcal{S}^! =< L \cup \{Fail\}, l0, V, V0, I, \Lambda, \rightarrow \cup \{(l, Fail, a(p), \bigwedge_{(l,l',a(p),G,A) \in \rightarrow} \neg G, (x := x)_{(x \in V)}) \mid l \in L, a(p) \in \Lambda^O\} >$

**Definition 3 (ioSTS product $\times$).**

The product of the ioSTS $\mathcal{S}_1 =< L_1, l0_1, V_1, V0_1, I_1, \Lambda_1, \rightarrow_1 >$ with the ioSTS $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$, denoted $\mathcal{S}_1 \times \mathcal{S}_2$, is the ioSTS $\mathcal{P} =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P} >$ such that $V_\mathcal{P} = V_1 \cup V_2$, $V0_\mathcal{P} = V0_1 \wedge V0_2$, $I_\mathcal{P} = I_1 \cup I_2$, $L_\mathcal{P} = L_1 \times L_2$, $l0_\mathcal{P} = (l0_1, l0_2)$, $\Lambda_\mathcal{P} = \Lambda_1 \cup \Lambda_2$. The transition set $\rightarrow_\mathcal{P}$ is the smallest set satisfying the following inference rules:

(1) $l_1 \xrightarrow{a(p),G_1,A_1}_{\mathcal{S}_1} l_2, l'_1 \xrightarrow{a(p),G_2,A_2}_{\mathcal{S}_2} l2' \vdash$
$(l_1, l'_1) \xrightarrow{a(p),G_1 \wedge G_2, A_1 \cup A_2}_{\mathcal{P}} (l_2, l'_2)$

(2) $l_1 \xrightarrow{a(p),G_1,A_1}_{\mathcal{S}_1} l_2, a(p) \notin \Lambda_2, l'_1 \in L_2 \vdash$
$(l_1, l'_1) \xrightarrow{a(p),G_1,A_1 \cup \{x:=x\}_{x \in V_2}}_{\mathcal{P}} (l_2, l'_1)$ (and symmetrically for $a(p) \notin \Lambda_1, l_1 \in L_1$)

### *3.2 Android applications and notations*

In this section, we define some notations to model intent-based behaviour of Android components with ioSTS. These notations are also given in Table 1.

To ease the writing of vulnerability patterns, we denote $AuthAct_{type}$ the action set that can be used with a type of component in accordance with the Android documentation. The APSET tool currently takes the types *Activity*, *Service*, *Activity* $\times$ *ContentProvider* and *Service* $\times$ *ContentProvider*. The two last types allows to model data vulnerabilities on Activities or Services composed with ContentProviders managing personal data.

Components communicates with intents, denoted with the action $intent(Cp, a, d, c, t, ed)$ with $Cp$ the called component, $a$ an action which has to be performed, $d$ a data expressed as a URI, $c$ a component category giving additional information about the action to execute, $t$ a type specifying the MIME type of the intent data and finally $ed$ which represent additional (extra) data [1]. Intent actions have different purposes, e.g., the action VIEW is given to an Activity to display something, the action PICK is called to choose an item and to return its URI to the calling component. Hence, we divided the action set, denoted $ACT$, into two categories: the set $ACT_r$ gathers the actions requiring the receipt of a response, $ACT_{nr}$ gathers the other actions. We also denote $C$, the set of predefined Android categories, $T$ the set of types. For instance, ?intent(ChooseActivity, PICK,content://com.android/con-tacts, DEFAULT„) represents the call of the activity ChooseActivity which lets a user choose a contact initially stored in the resource "content://com.android/
contacts".

| Notation | Meaning |
|---|---|
| $AuthAct_{type}$ | action set of a component type |
| $ACT_r$ | set of intent actions requiring a response |
| $ACT_{nr}$ | set of intent actions which do not require a response |
| $?intent(Cp, a, d, c, t, ed)$ | intent composed of: called component $Cp$, action $a$, data $d$, action category $c$, data type $t$, extra data $ed$ |
| $C$ | set of Android categories |
| $T$ | set of Android types |
| $URI$ | set of URI found in an application completed with random URIs |
| $RV$ | set of predefined and random values |
| $INJ$ | set of SQL and XML injections |
| !Display | display of a screen by an Activity |
| !Running | Service under execution in the background |
| !ComponentExp | Exception raised by a component |
| !SystemExp | Exception raised by the system |
| $?call(Cp, requ est, tableURI)$ | ContentProvider call with request on the table tableURI |
| $!callResp(Cp, resp)$ | ContentProvider response with $resp$ the content of the response |

**Table 1.** Android component notations

Android components may raise exceptions that we group into two categories: those raised by the Android system on account of the crash of a component and the other ones. This difference can be observed while testing with our framework. This is respectively modelled with the actions *!SystemExp* and *!ComponentExp*. Android components reply to intents with different actions in reference to their types. Activities, which are the most common Android components, display screens to let users interact with programs. We denote $!Display(\ A)$ the action modelling the display of a screen for the Activity $A$. Services are executed in background and usually aim to return data. This is denoted with the action *!Running(A)*.

With these notations, one can deduce that $AuthAct_{Activity}$ is the set $\{?intent(Cp, a, d, c, t, ed), !Display(\ A), !SystemExp, !ComponentExp, !\delta\}$. $AuthAct_{Service}$

gathers the actions $\{?intent(Cp, a, d, c, t, ed), !Running(A), !SystemExp, !ComponentExp, !\delta\}$.

ContentProviders are components whose functioning is slightly different. ContentProviders do not receive intents but SQL-oriented queries that we denote $?call(Cp, request, tableURI)$ with $Cp$, the called ContentProvider and $request$, the query over the table $table - URI$. A response, denoted $!callResp(Cp, cursor)$, is possibly returned. Consequently, $AuthAct_{ContentProvider}$ is the set $\{?call(Cp, request, tableURI), !callResp(Cp, cursor), !\delta, !Co$ $!SystemExp\}$.

Any action defined in this section, has its corresponding function coded in the tool. For instance, the action $!Display(A)$ is coded by the function Display() returning true if a screen is displayed. This link between actions and Java code makes easier the construction of final test cases, which actually call Java sections of code that can be executed. Naturally, this action set can be upgraded.

### *3.3 Vulnerability patterns and vulnerability status*

Rather than defining the vulnerabilities of a specification, (which have to be written for each specification), we chose defining vulnerability patterns for describing intent-based vulnerabilities of an Android component type. In general terms, a vulnerability pattern describes a widespread vulnerability scenario that can be experimented on several components. If required, a vulnerability pattern can still be specialised to a specific component though.

A vulnerability pattern, denoted $\mathcal{V}$, is modelled with a specialised ioSTS suspension. $\mathcal{V}$ has to be equipped of actions used for describing Android components. Consequently, the action set $\Lambda_{\mathcal{V}}$ of $\mathcal{V}$ has to be equal to $AuthAct_{type}$.

A vulnerability pattern is also composed of two distinct final locations *Vul*, *NVul* which aim to recognise the vulnerability status over component executions. Intuitively, vulnerability pattern paths starting from the initial location and ended by *Vul*, describe the presence of a vulnerability. By deduction, paths ended by *NVul* express functional behaviours which show the absence of the vulnerability. $\mathcal{V}$ is also output-complete to recognise a status whatever the actions observed while testing. Transition guards are composed of specific predicates to falicitate their writing. In the paper, we consider some predicates such as *in*, which stands for a Boolean function returning true if a parameter list belongs to a given value set or *streq* which returns true if two String values are equals. Furthermore, we consider several value sets to categorise malicious values and attacks: *RV* is a set of values known for relieving bugs enriched with random values. *INJ* is a set gathering XML and SQL injections constructed from database table URI found in the tested Android application. *URI* is a set gathering the URI found in the tested Android application completed with URI randomly constructed from the previous

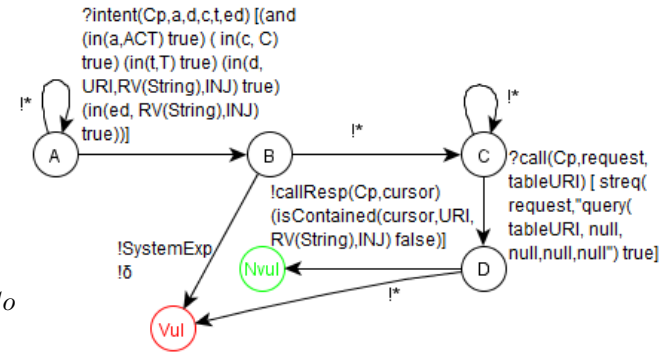


**Fig. 6.** Vulnerability pattern example

ones. ***These sets, written in XML, can be edited by end-users, for instance in reference to values provided by the OWASP organisation. . completed automatically by the tools with values found in the Android application code ?????***New sets can also be referred in vulnerability patterns upon condition that real value sets with the same names would be added to APSET.

**Definition 4 (Vulnerability pattern).**

A vulnerability pattern is a deterministic and output-complete ioSTS suspension $\mathcal{V} =< L_{\mathcal{V}}, l0_{\mathcal{V}}, V_{\mathcal{V}}, V0_{\mathcal{V}}, I_{\mathcal{V}}, \Lambda_{\mathcal{V}}, \rightarrow_{\mathcal{V}}>$ composed of sink locations in $L_{\mathcal{V}}$ which belong to $\{Vul, NVul\}$. $type(\mathcal{V})$ is the component (or component composition) type targeted by $\mathcal{V}$. The action set $\Lambda_{\mathcal{V}} = AuthAct_{type}$ where $type$ is equal to $type(\mathcal{V})$.

*Example 2.* Figure 6 illustrates a straightforward example of vulnerability pattern, related to data integrity. It aims to check whether an Activity, called with intents composed of malicious data, cannot alter the content of a database table managed by a ContentProvider. For readability, the label $!*$ is a shortcut notation for all valued output actions that are not explicitly labelled by other transitions. Guards are written with the SMT-LIB language as in APSET. Intents are constructed with data and extra data composed of malformed URI or String values known for relieving bugs or XML/SQL injections. If the called component crashes (!SystemExp) or is quiescent ($!\delta$), it is considered as vulnerable. Once the intent is performed, the ContentProvider is called with the *query* function of the Android SDK to retrieve all the data stored in a table whose URI is given by the variable *tableURI*. If the result set is not composed of test data given in the intent (checked with the predicate isContained), then the component is not vulnerable. Otherwise, it is vulnerable.

As stated previously, the sink locations $Vul$ and $NVul$ of a vulnerability pattern help recognise the status of an ioSTS. Given an ioSTS $\mathcal{S}$, its status can be checked w.r.t. a vulnerability pattern $\mathcal{V}$ if and only if both are compatible, i.e., they share the same action set, the same parameters and have distinct internal variables:

**Definition 5 (Compatible ioSTS).** An ioSTS $\mathcal{S}_1 = < L_1, l_1^0, V_1, V_1^0, I_1, \Lambda_1, \rightarrow_1 >$ is compatible with $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$ iff

- $\Lambda_1^I = \Lambda_2^I$, $\Lambda_1^O = \Lambda_2^O$,
- $V_1 \cap V_2 = \emptyset$, $I_1 = I_2$.

The vulnerability status of an ioSTS $\mathcal{S}$ is defined over its observable valued action sequences. The latter are called traces and are extracted from ioLTS semantics:

**Definition 6 (Runs and traces).**
For an ioSTS $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow >$, interpreted by its ioLTS semantics $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \rightarrow >$, a run $q_0\alpha_0...\alpha_{n-1}q_n$ is an alternate sequence of states and valued actions. $Run(\mathcal{S}) = Run([\![\mathcal{S}]\!])$ is the set of runs found in $[\![\mathcal{S}]\!]$. $Run_F(\mathcal{S})$ is the set of runs of $\mathcal{S}$ finished by a state in $F \times D_V \subseteq Q$, with $F$ a location set in $L$.

It follows that a trace of a run $r$ is defined as the projection $proj_{\sum}(r)$ on actions. $Traces_F(\mathcal{S}) = Traces_F([\![\mathcal{S}]\!])$ is the set of traces of all runs finished by states in $F \times D_V$.

For example, the trace $?intent(ChooseActivity, PICK, content$ $//com.android/contacts, DEFAULT, , ))!\delta \in Traces_{Vul}(\mathcal{V})$ is a trace of the vulnerability pattern $\mathcal{V}$ of Figure 6, leading to the $Vul$ location. Now, if this trace is observed from the Android component ChooseActivity, then it reveals that the latter is vulnerable to $\mathcal{V}$. Intuitively, Trace sets can be employed to define vulnerability status:

**Definition 7 (Vulnerability status of an ioSTS).**
Let $\mathcal{S}$ be an ioSTS, $\mathcal{V}$ be a vulnerability pattern such that $\mathcal{S}^\delta$ is compatible with $\mathcal{V}$. We define the vulnerability status of $\mathcal{S}$ (and of its underlying ioLTS semantics $[\![\mathcal{S}]\!]$) over $\mathcal{V}$ with:

- $\mathcal{S}$ is not vulnerable to $\mathcal{V}$, denoted $\mathcal{S} \models \mathcal{V}$ if $Traces(\mathcal{S}^\delta) \subseteq Traces_{NVul}(\mathcal{V})$,
- $\mathcal{S}$ is vulnerable to $\mathcal{V}$, denoted $\mathcal{S} \not\models \mathcal{V}$ if $Traces(\mathcal{S}^\delta) \cap Traces_{Vul}(\mathcal{V}) \neq \emptyset$.

We assume here that vulnerability patterns are correctly modelled and express vulnerabilities. On the contrary, false positive status may appear, but these ones do not reflect the detection of vulnerabilities. Now that we have a vulnerability pattern modelling language and a vulnerability pattern status definition, we are ready to detail the functioning of the security testing method implemented in APSET.

## 4 Testing methodology

Assuming that we have a set of vulnerability patterns modelled with ioSTS suspensions, we describe, in the following, the automatic test case generation and execution based upon the generation of partial specifications from an Android application.

### *4.1 Partial specification generation (part A in Figure 1)*

Directly experimenting Android components with vulnerability patterns would lead to several issues. For example, an Activity which displays screens on a smartphone has a different purpose than a Service which does not interact directly with a user. Performing a kind of blind testing without considering the component features and type would often lead to false positive results. Besides, Android applications and the Android documentation gather a lot of information that can be used to produce partial models:

1. from an Android application packaged in a .apk file, APSET respectively calls the tool *dextojar* to produce a .jar package and the tool *apktool* to extract the Manifest file. A simplified class diagram, depicting the Android components of the application and their types, is initially computed from the .jar package. Class methods and attribute names are retrieved by applying reverse engineering based on Java reflection. This class diagram also establishes the relationships among components. In particular, we detect the compositions of an Activity or a Service *ct* with a ContentProvider *cp*. This relationship is established when a component has a *Contentresolver* attribute. From this diagram, we extract the list $L_C$, composed of the products $ct \times cp$ and the list of Activities and Services of the application,
2. a partial specification $\mathcal{S}_{comp} = (S1^!_{comp}, S2^!_{comp})$ is generated for each component *comp* (or component composition) in $L_C$. $S1_{comp}$ is an ioSTS suspension generated from the implicit intents given in the Manifest file of the application. In contrast, $S2_{comp}$ is built with the explicit intents. Both ioSTS are completed on the output set to produce $(S1^!_{comp}, S2^!_{comp})$. This separation shall be particularly useful to distribute the test case set between implicit and explicit intents when the number of test cases is limited.
The tuple $(S1_{comp}, S2_{comp})$ is constructed two with algorithms based upon some knowledge of the Android documentation. For sake of readability, Algorithms 1 and 2 are simplified versions of those used in APSET. If *comp* is an Activity (resp. a Service), Algorithm 1 (resp. Algorithm 2) constructs ($S1_{comp}$, $S2_{comp}$) from the intent filters *IntentFilter(act,cat, data)* found in the Manifest file *MF*. The action sets of $\Lambda_{Si_{comp}}(i = 1, 2)$ are equal to $AuthAct_{type(comp)}$ with *type*(*comp*) the type of the component, e.g., Activity. Algorithm 1 produces two ioSTS w.r.t. the intent functioning described in the Android documentation. Firstly, Algorithm 1 constructs $S1_{comp}$ with the implicit intents found in the intent filters (lines 6-16). Depending on the action type read, the guard of the output action is completed to reflect the fact that a response may be received or not. If the action

of the intent filter is unknown (lines 13,14), no guard is formulated on the output action (a response may be received or not). While the generation of $S1_{comp}$, the algorithm also produces a guard $G$ equals to the negation of the union of guards carried by the constructed ?intent actions (line 16). Afterwards, $S2_{comp}$ is constructed by means of this guard: it models the sending of an intent with the guard $G$ (intuitively, any intent except the intents of $S1_{comp}$) followed by a transition carrying the action !Display without guard and a transition labelled by !ComponentExp. If the component is a Service, its partial specification is obtained with Algorithm 2 whose functioning is similar (except for its action set).

If $comp = ct \times cp$ is a composition of an Activity or a Service $ct$ with a ContentProvider $cp$, the generation of $(S1_{comp}, S2_{comp})$ is slightly different. As previously, a tuple $(S1_{ct}, S2_{ct})$ is produced either with Algorithm 1 or Algorithm 2. $(S1_{comp}, S2_{comp})$ is constructed with the products $Si_{ct} \times S_{cp} (i = 1, 2)$ where $S_{cp}$ is an ioSTS suspension modelling the call of the ContentProvider $cp$. $S_{cp}$ is derived from a generic ioSTS where only the ContentProvider name and the variable tableURI are updated from the information found in the Manifest file. An example is depicted in Figure 8 for a ContentProvider, named Contacts, managing contact information with the table "ContactsContract.RawContacts". Naturally, this specification is written in accordance with the set $AuthAct_{ContentProvider}$.

Finally, the partial specification $\mathcal{S}_{comp} = (S1^{!}_{comp}, S2^{!}_{comp})$ is achieved by completing both $S1_{comp}$ and $S2_{comp}$ on the output set with new transitions to the $Fail$ location. The latter shall be particularly useful to refine the test verdict by helping recognise correct and incorrect behaviours of an Android component w.r.t. its specification.

Correctness and completeness of Algorithms 1 and 2: both algorithms are very similar, they mainly differ from the produced actions. Hence, we consider Algorithm 1 below. Firstly, if no intent filter is found in the Manifest file, $S1_{comp}$ is empty. Then, $S2_{comp}$ is composed of 3 transitions expressing that the component accepts any explicit intent and returns either a screen or returns an exception (not a crash). This corresponds exactly to the definition of an Activity as referred in the Android documentation. If it exists at least one intent filter, then $S1_{comp}$ is composed of a transition expressing the receipt of the corresponding intent followed by transitions carrying an action related to the intent action found in the intent filter. The behaviour related to this intent filter is removed from $S2_{comp}$. All the intent actions found in the Android documentation are categorised in the sets $ACT_nr$ and $ACT_r$. Hence, Algorithm 1 produces ioSTS that correctly express the intent related behaviours of Activities. The algorithm is also complete since it can take any Manifest file (all the actions of the Android documentation are supported).

Both algorithms are linear in time since they produce some transitions for the finite intent filter set found in the Manifest file.

```
1 <activity class=".A"
      android:label="@string/title_A">
3             <intent-filter>
                  <action android:name="android.
                      intent.action.PICK" />
5                 <category android:name="android.
                      intent.category.DEFAULT" />
                  <data android:mimeType="content:
                      //com.android.contacts" />
7             </intent-filter>
  </activity>
```

**Fig. 7.** The intent filter section of the Manifest file related to the Activity A

**Algorithm 1:** Partial Specification Generation for Activities

**input** : Manifest file *MF*, Activity *comp*
**output**: Tuple $(S1_{comp}, S2_{comp})$

1 $A_i$ is the identity assignment $(x := x)_{x \in V_{Si_{comp}}} (i = 1, 2)$;
2 $it := 0; G := \emptyset$;
3 $\Lambda_{Si_{comp}} (i = 1, 2) = AuthAct_{type(comp)}$;
4 Add $(l0_{Si_{comp}}, l0_{Si_{comp}}, !\delta, , )$ to $\rightarrow_{Si_{comp}}$ $(i = 1, 2)$;
5 **foreach** *IntentFilter(act,cat,data) of comp in MF* **do**
6 $\quad it := it + 1$ ;
7 $\quad$ Add $(l0_{S1_{comp}}, l_{it,1}, ?intent(Cp, a, d, c, t, ed)$, $[(and\ \ (streq(Cp, comp)\ true) G1)], A_1)$ to $\rightarrow_{S1_{comp}}$ with $G1 : (and\ (streq(a, act)\ true)(streq(d, data)\ true)\ (streq(c, cat)\ true))$;
8 $\quad$ **if** $act \in ACT_r$ **then**
9 $\quad\quad$ Add $(l_{it,1}, l0_{S1_{comp}}, !Display(comp)$, $[(streq(comp.resp,\ null)\ false)], A_1)$ to $\rightarrow_{S1_{comp}}$;
10 $\quad$ **else if** $act \in ACT_{nr}$ **then**
11 $\quad\quad$ Add $(l_{it,1}, l0_{S1_{comp}}, !Display(comp)$, $[(streq(comp.resp, null)\ true)], A_1)$ to $\rightarrow_{S1_{comp}}$
12 $\quad$ **else**
13 $\quad\quad$ Add $(l_{it,1}, l0_{S1_{comp}}, !Display(comp), , A_1)$ to $\rightarrow_{S1_{comp}}$;
14 $\quad$ Add $(l_{it,1}, l0_{S1_{comp}}, !ComponentExp, , )$ to $\rightarrow_{S1_{comp}}$;
15 $\quad G := G \wedge \neg G1$;
16 Add $(l0_{S2_{comp}}, l_1, ?intent(Cp, a, d, c, t, ed)$, $[(and\ (streq(Cp, comp)\ true) G)], A_2)$, $(l_1, l0_{S2_{comp}}, !Display(comp), , A_2)$, $(l_1, l0_{S2_{comp}}, !ComponentExp\ , , )$ to $\rightarrow_{S2_{comp}}$;

**Algorithm 2:** Partial Specification Generation for Services

**input** : Manifest file $MF$, Service $comp$
**output**: Tuple $(S1_{comp}, S2_{comp})$

$A_i$ is the identity assignment $(x := x)_{x \in V_{Si_{comp}}} (i = 1, 2)$;
$it := 0; G := \emptyset$;
$\Lambda_{Si_{comp}} (i = 1, 2) = AuthAct_{type(comp)}$;
Add $(l0_{Si_{comp}}, l0_{Si_{comp}}, !\delta, , )$ to $\rightarrow_{Si_{comp}}$ $(i = 1, 2)$;
**foreach** *IntentFilter(act,cat,data) of comp in MF* **do**
  $it := it + 1$ ;
  Add $(l0_{S1_{comp}}, l_{it,1}, ?intent(Cp, a, d, c, t, ed), [(and \ \ (streq(Cp, comp) \ true) G1)], A_1)$ to $\rightarrow_{S1_{comp}}$ with $G1 : (and \ (streq(a, act) \ true) \ (streq(d, data) \ true) \ (streq(c, cat) \ true))$;
  **if** $act \in ACT_r$ **then**
    Add $(l_{it,1}, l0_{S1_{comp}}, !Running(comp), [(streq(comp.resp, null) \ false)], A_1)$ to $\rightarrow_{S1_{comp}}$;
  **else**
    Add $(l_{it,1}, l0_{S1_{comp}}, !Running(comp), , A_1)$ to $\rightarrow_{S1_{comp}}$;
  Add $(l_{it,1}, l0_{S1_{comp}}, !ComponentExp, , )$ to $\rightarrow_{S1_{comp}}$;
  $G := G \wedge \neg G1$;
Add $(l0_{S2_{comp}}, l_1, ?intent(Cp, a, d, c, t, ed), [(and \ (streq(Cp, comp) \ true) G)], A_2)$, $(l_1, l0_{S2_{comp}}, !Running(comp), , A_2)$, $(l_1, l0_{S2_{comp}}, !ComponentExp , , )$ to $\rightarrow_{S2_{comp}}$;

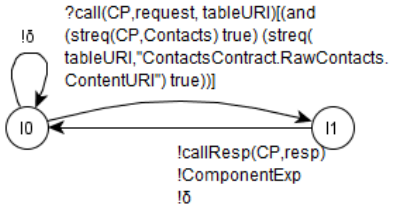


**Fig. 8.** ContentProvider specification

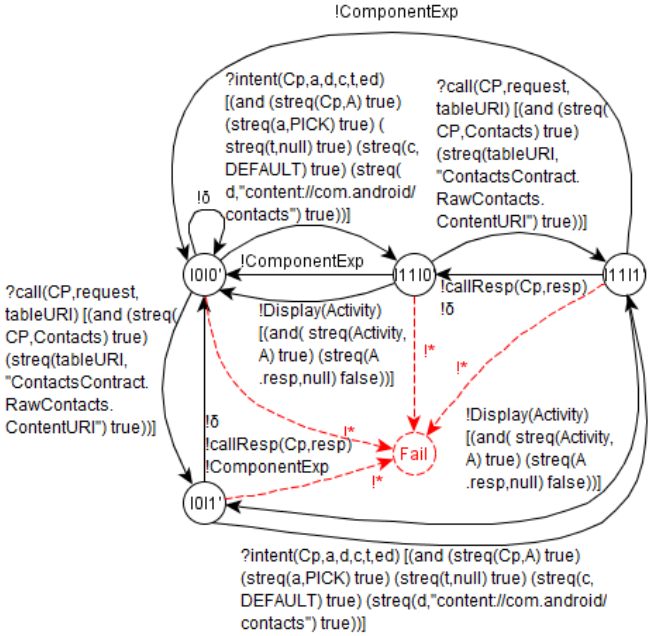


**Fig. 9.** A specification example

*Example 3.* Figure 9 illustrates the resulting ioSTS suspension $S1^!_{comp}$ which stems from the product of one Activity with the previous Contact-Provider depicted in Figure 8. The part of the Manifest related to this activity is depicted in Figure 7. The Activity accepts intents composed of the PICK action and data whose URI corresponds to the contact list stored in the device. Basically, this Activity aims to return the contact chosen by a user in its contact list. The resulting ioSTS composition also accepts requests to the Contact-Provider. Incorrect behaviour, e.g., the display of a screen without the receipt of response, are expressed with transitions to Fail.

As for vulnerability patterns, one can now define the status of an ioSTS $\mathcal{S}$ with a partial specification $\mathcal{S}_{comp}$. We believe it would not be appropriate to discuss about conformance or compliance with the Android documentation since a non exhaustive part of this documentation appears across $\mathcal{S}_{comp}$. This is why we chose to talk about non-compliance:

**Definition 8 (Compliance status).**
Let $\mathcal{S}$ be an ioSTS and $\mathcal{S}_{comp}$ be a partial specification such that $\mathcal{S}^\delta$ is compatible with $\mathcal{S}_{comp}$. We define the non compliance of $\mathcal{S}$ (and of its underlying ioLTS semantics $[\![\mathcal{S}]\!]$) over $\mathcal{S}_{comp}$ with:

$\mathcal{S}$ does not comply with $\mathcal{S}_{comp}$, denoted $\mathcal{S} \not\vDash \mathcal{S}_{comp}$ if $Traces(\mathcal{S}^\delta) \cap (Traces_{Fail}(S1^!_{comp}) \cup Traces_{Fail}(S2^!_{comp})) \neq \emptyset$.

For example, if we assume that the ioSTS $\mathcal{S}^\delta$ is compatible with the partial specification $S1^!_{comp}$ of Figure 9 and if $?intent(ChooseActivity, PICK, content : //com.android/co$ $null)$ is a trace of $\mathcal{S}^\delta$, then $\mathcal{S}$ does not comply with this specification since this trace belongs to $Traces_{Fail}(S1^!_{comp})$.

### *4.2 Test case selection (part B in Figure 1)*

Test cases stem from the combination of vulnerability patterns with compatible partial specifications. This combination is defined here with a parallel composition.

The parallel composition of two ioSTS is a specialised product which illustrates the shared behaviour of two original ioSTS that are compatible:

**Definition 9 (Parallel composition $||$).**
The parallel composition of two compatible ioSTS $\mathcal{S}_1, \mathcal{S}_2$, denoted $\mathcal{S}_1 || \mathcal{S}_2$, is the ioSTS $\mathcal{P} =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P} >$ such that $V_\mathcal{P} = V_1 \cup V_2$, $V0_\mathcal{P} = V0_1 \wedge V0_2$, $I_\mathcal{P} = I_1 \cup I_2$, $L_\mathcal{P} = L_1 \times L_2$, $l0_\mathcal{P} = (l0_1, l0_2)$, $\Lambda_\mathcal{P} =$

$\Lambda_1 \cup \Lambda_2$. The transition set $\rightarrow_{\mathcal{P}}$ is the smallest set satisfying the first rule of Definition 3.

**Lemma 1 (Parallel composition traces).**
*If $\mathcal{S}_2$ and $\mathcal{S}_1$ are compatible then $Traces_{F_1 \times F_2}(\mathcal{S}_1 || \mathcal{S}_2) = Traces_{F_1}(\mathcal{S}_1) \cap Traces_{F_2}(\mathcal{S}_2)$, with $F_1 \subseteq L_{\mathcal{S}_1}$, $F_2 \subseteq L_{\mathcal{S}_2}$.*

Given a vulnerability pattern $\mathcal{V}$ and a partial specification $\mathcal{S}_{comp} = (S1^!_{comp}, S2^!_{comp})$, the composition $\mathcal{VP}_{comp} = (\mathcal{V}||S1^!_{comp}, \mathcal{V}||S2^!_{comp})$ is called a vulnerability property of $\mathcal{S}_{comp}$. These parallel compositions $(\mathcal{V}||Si_{comp})(i = 1, 2)$ produce new locations and in particular new final verdict locations:

**Definition 10 (Verdict location sets).**
Let $\mathcal{V}$ be a vulnerability pattern and $\mathcal{S}_{comp} = (S1^!_{comp}, S2^!_{comp})$ a partial specification such that $Si_{comp}(i = 1, 2)$ are compatible with $\mathcal{V}$. $(\mathcal{V}||Si^!_{comp})(i = 1, 2)$ are composed of new locations recognising vulnerability status:

1. **NVUL** $= NVul \times L_{Si^!_{comp}}$. **NVUL/FAIL** $= (NVul, Fail) \in NVUL$ aims to recognise non-compliance w.r.t. the partial specification $\mathcal{S}_{comp}$ and the non-vulnerable status w.r.t. $\mathcal{V}$,
2. **VUL** $= Vul \times L_{Si^!_{comp}}$. **VUL/FAIL** $= (Vul, Fail)$ aims to recognise non-compliance w.r.t. $\mathcal{S}_{comp}$ and the vulnerable status w.r.t. $\mathcal{V}$,
3. **FAIL** $= L_{\mathcal{V}} \times Fail$ recognises incorrect behaviour w.r.t. $\mathcal{S}_{comp}$.

A vulnerability property only keeps the shared behaviour of vulnerability pattern and of a partial specification to later produce executable test cases. If a vulnerability pattern is inconsistent with a component (incorrect actions, etc.), the parallel composition of the vulnerability pattern with the component specification gives paths ended by locations that are not verdict locations. In this case, the test case generation is stopped.

Thereafter, test cases are achieved with Algorithm 3 which performs the two following main steps on the vulnerability property $V(\mathcal{S}_{comp}) = (\mathcal{V}||S1^!_{comp}, \mathcal{V}||S2^!_{comp})$. Firstly, it splits $(\mathcal{V}||Si^!_{comp})(i = 1, 2)$ into several ioSTS. Intuitively, from a location $l$ having $k$ transitions carrying an input action, e.g., an intent, $k$ new test cases are constructed to experiment the component under test with the $k$ input actions and so on for each location $l$ having transitions labelled by input actions (lines 2-5). A set of valuation tuples is computed from the list of undefined parameters found in the input action (line 6). For instance, intents are composed of several variables whose domains are given in guards. These ones have to be concretised to obtain executable test cases (i.e. each undefined parameter of an input action is assigned to a value). Instead of using a cartesian product to construct a tuple of valuations, we adopted a Pairwise technique [8]. Assuming that errors can be revealed by modifying pairs of variables, this technique strongly reduces the coverage of variable domains by constructing discrete combinations for pair of parameters only. The set of valuation tuples is constructed with the *Pairwise* procedure which takes the list of undefined parameters and the transition guard to find the parameter domain. If no domain is found, the $RV$ set is used instead. In the second step (line 7-14), input actions are concretised. Given a transition $t$ and its set of valuation tuples $P(t)$, this step constructs a new test case for each tuple $pv = (p_1 = v_1, ..., p_n = v_n) \in P(t)$ by replacing the guard $G$ with $G \wedge pv$ iff $G \wedge pv$ is satisfiable. Finally, if the resulting ioSTS suspension $tc$ has verdict locations, then $tc$ is added into the test case set $TC_{(\mathcal{V}||Si^!_{comp})}$. Steps 1. and 2. are iteratively applied until each combination of valuation tuples and each combination of transitions labelled by input actions are covered. This algorithm may produce a wide set of test cases, e.g., if the number of tuple of values given by the Pairwise function is large. This is why we added the end condition of lines (18,19) to limit the test case number but also to balance the generation of test cases built with implicit intents (those obtained from $S1^!_{comp}$) with the test cases executing explicit intents (obtained from $S2^!_{comp}$).

A test case example, derived from the specification of Figure 9 and the vulnerability pattern of Figure 6 is depicted in Figure 10. This one is commented below.

Correctness and completeness of Algorithm 3: here we assume that vulnerability patterns are correctly modelled and composed of Vul and NVul locations. Since it is assumed that the vulnerability $\mathcal{V}$ and the partial specification $S_comp$ are compatible, Algorithm 3 takes vulnerability properties that are composed of input actions (?intent) and output actions (at least actions modelling exceptions) and of locations in VUL and NVUL. Thus, Algorithm 3 can concretise the input actions and can extract test cases from these vulnerability properties.
The test selection in Algorithm 3 is correct iff the test case traces belong to the trace set of the vulnerability property. This is captured by the following Proposition:

**Proposition 1.** *Let $\mathcal{VP}_{comp}$ be a vulnerability property derived from the composition of a vulnerability pattern $\mathcal{V}$ and a partial specification $\mathcal{S}_{comp} = (S1^!_{comp}, S2^!_{comp})$. $TC$ is the test case set generated by Algorithm 3. We have $\forall tc \in TC$, $Traces(tc) \subseteq (Traces(\mathcal{V}||S1^!_{comp}) \cup Traces(\mathcal{V}||S2^!_{comp}))$.*

Intuitively, in step 1, Algorithm 3 selects each input action (one after one) from a location having several outgoing transitions carrying input actions. It builds test cases composed of $Si^!_{comp}$ paths starting from its initial location. In step2, it concretises each transition labelled by an input action with values satisfying its guard and checks that the resulting transition can be reached from the initial state of the $Si^!_{comp}$. In short, the ioLTS semantics of the resulting test case is only composed of paths

**Algorithm 3:** Test case generation

**input** : A vulnerability property $\mathcal{VP}_{comp}$, *tcnb* the maximal number of test cases
**output**: Test case set $TC$

**foreach** $\mathcal{V}||Si^!_{comp}(i=1,2) \in \mathcal{VP}_{comp}$ **do**
  *Step 1. input action choice*
  **foreach** *location $l$ having outgoing transitions carrying input actions* **do**
    Choose a transition $t = l \xrightarrow{?a(p),G,A}_{\mathcal{V}||Si^!_{comp}} l_2$;
    remove the other transitions labelled by input actions;
    $P(t) := Pairwise(p_1, ..., p_n, G)$ with $(p_1, ..., p_n) \subseteq p$ the list of undefined parameters;
  *Step 2. input concretisation*
  $G' := V0_{\mathcal{V}||Si^!_{comp}}$;
  Call $Reach(l0_{\mathcal{V}||Si^!_{comp}}, G')$;
  $Reach(l, G')$:
  **begin**
    **foreach** $t = l \xrightarrow{a(p),G,A}_{\mathcal{V}||Si^!_{comp}} l_2$ **do**
      **if** $a(p)$ *is an input* **then**
        Choose a valuation tuple $pv = (p_1 = v_1, ..., p_n = v_n)$ in $P(t)$;
        **if** $G' \wedge G \wedge pv$ *is satisfiable* **then**
          Replace $G$ by $G \wedge pv$ in $t$;
          $Reach(l_2, G' \wedge G \wedge pv)$
        **else**
          Choose another valuation tuple in $P(t)$;
      **else**
        $Reach(l_2, G' \wedge G)$ ;
  $tc$ is the resulting ioSTS suspension;
  *Step 3.*
  **if** *$tc$ has reachable verdict locations in* VUL *and in* NVUL **then**
    $TC_{\mathcal{V}||Si^!_{comp}} := TC_{\mathcal{V}||Si^!_{comp}} \cup \{tc\}$ ;
  **if** $Card(TC_{\mathcal{V}||Si^!_{comp}}) \geq tcnb$ **then**
    STOP;
Repeat 1. and 2. until each each combination of valuation tuples and combination of transitions carrying input actions are covered;
$TC = \bigcup_{i=1,2} TC_{(\mathcal{V}||Si^!_{comp})}$;

of the ioLTS semantics of $Si^!_{comp}$. As a consequence, the test case traces belong to the trace set of the vulnerability property.

Furthermore, Algorithm 3 takes finite time to compute test case sets. Indeed, step 1 constructs set of values for each input action of $Si^!_{comp}$ with a Pairwise technique which is logarithmic in time. In the second step, transitions of $Si^!_{comp}$ are covered at most *tcnb* times (*tcnb* is the maximum test case number). Each time an input action is concretised, the new guard satisfiability is checked with a solver whose complexity should be bounded (this complexity depends on the solver choice).

### *4.3 Test case execution*

A component under test ($CUT$) is considered as a black box whose only interfaces are known. However, one usually assumes the following test hypotheses to execute test cases:

- the functional behaviour of the component under test, observed while testing, can be modelled by an ioLTS $CUT$. $CUT$ is unknown (and potentially nondeterministic). $CUT$ is assumed input-enabled (it accepts any of its input actions from any of its states). $CUT^\delta$ denotes its ioLTS suspension,
- to be able to dialog with $CUT$, one assumes that $CUT$ is a component whose type is the same as the component type targeted by the vulnerability pattern $\mathcal{V}$ and that it is compatible with $\mathcal{V}$.

Thanks to these assumptions, Definition 7 now captures that the vulnerability status of $CUT$ against a vulnerability pattern $\mathcal{V}$ can be determined with the $CUT$ traces. These are constructed by experimenting $CUT$ with test cases. Usually, the execution of test cases is defined by their parallel compositions with the implementation:

**Proposition 2 (Test case execution).**
*Let $TC$ be a test case set obtained from the vulnerability pattern $\mathcal{V}$. $CUT$ is the ioLTS of the component under test, assumed compatible with $\mathcal{V}$. For all test case $tc \in TC$, the execution of $tc$ on $CUT$ is defined by the parallel composition $tc||CUT^\delta$.*

*Remark 1.* A test case $tc$ obtained from a vulnerability pattern $\mathcal{V}$, can be experimented on $CUT$ since $tc$ and $CUT$ are compatible. Indeed, $tc$ is produced from a vulnerability property $\mathcal{VP}_{comp} = (\mathcal{V}||S1^!_{comp}, \mathcal{V}||S2^!_{comp})$ such that $\Lambda_{Si^!_{comp}}(i=1,2) = AuthAct_{type(comp)}$ (Algorithm 3). The action set of $\mathcal{V}$ is also equal to $AuthAct_{type(comp)}$. We deduce that $tc$ is compatible with $\mathcal{V}$, which is also compatible with $CUT$ (test hypothesis).

The above proposition leads to the test verdict of a component under test against a vulnerability pattern $\mathcal{V}$. Originally, this one refers to the vulnerability status definition and is completed by the detection of incorrect behaviour described in partial specifications with the verdict locations *VUL/FAIL* and *NVUL/FAIL*. An inconclusive verdict is also defined when a FAIL verdict location is reached after a test case execution. This verdict means that incorrect actions or data were received. To avoid false positive results, the test is stopped without completely executing the scenario given in the vulnerability pattern.

**Definition 11 (Test verdict).**

Let $TC$ be a test case set obtained from the vulnerability pattern $\mathcal{V}$ and the partial specification $\mathcal{S}_{comp} = (S1^!_{comp}, S2^!_{comp})$ (with $Si_{comp}(i = 1, 2)$ compatible with $\mathcal{V}$). $CUT$ is the ioLTS of the component under test, assumed compatible with $\mathcal{V}$.

The execution of the test case set $TC$ on $CUT$ yields one of the following verdicts:

1. $VUL$ iff $\exists tc \in TC$, $tc||CUT^\delta$ produces a trace $\sigma$ such that $\sigma \in Traces_{VUL}(tc)$ and $\forall tc2 \neq tc \in TC$, $tc2||CUT^\delta$ produces $\sigma \notin Traces_{VUL/FAIL}(tc2)$,
2. $VUL/FAIL$ iff $\exists tc \in TC$, $tc||CUT^\delta$ produces $\sigma \in Traces_{VUL/FAIL}(tc)$,
3. $NVUL$ $\forall tc \in TC, tc||CUT^\delta$ produces $\sigma \in Traces_{NVUL}(t$
4. $NVUL/FAIL$, iff $\exists tc \in TC$, $tc||CUT^\delta$ produces $\sigma \in Traces_{NVUL/FAIL}(tc)$ and $\forall tc2 \neq tc \in TC, tc2||CUT^\delta$ produces $\sigma \in Traces_{NVUL}(tc2) \cup Traces_{NVUL/FAIL}(tc2)$.
5. $Inconclusive$, iff $\exists tc \in TC, tc||CUT^\delta$ produces $\sigma \in Traces_{FAIL}(tc)$.

**Proposition 3 (Test verdict correctness).**

*With the notations of Proposition 2 and Definition ??, we assume having a test case set $TC$ that returns a verdict $v$ to a component $CUT$. Consequently,*

1. *if $v = VUL$, then $CUT$ is vulnerable to $\mathcal{V}$,*
2. *if $v = VUL/FAIL$, then $CUT$ is vulnerable to $\mathcal{V}$ and $CUT$ does not also respect the component normal functioning expressed in $\mathcal{S}_{comp}$,*
3. *if $v = NVUL$, then, $CUT$ is not vulnerable to $\mathcal{V}$,*
4. *if $v = NVUL/FAIL$, then, $CUT$ is not vulnerable to $\mathcal{V}$. However, $CUT$ does not respect the component normal functioning expressed in $\mathcal{S}_{comp}$,*
5. *if $v = Inconclusive$, then $CUT$ has an unknown status.*

Sketch of proof of 1 and 2:
$\exists tc \in TC$ such that $tc||CUT^\delta$ produces a trace $\sigma \in Traces_{VUL}(tc)$.
Thus, $Traces_{VUL}(tc) \cap Traces(CUT^\delta) \neq \emptyset$ (Lemma 1)
$(Traces_{VUL}(\mathcal{V}||S1^!_{comp}) \cup Traces_{VUL}(\mathcal{V}||S2^!_{comp})) \cap Traces(CUT^\delta) \neq \emptyset$ (Proposition 1)
$Traces_{VUL}(\mathcal{V}||Si^!_{comp}) = Traces_{Vul}(\mathcal{V}) \cap Traces_{L_{Si^!_{comp}}}(Si^!_{comp})$ since $Si^!_{comp}$ is compatible with $\mathcal{V}$ (Algorithm 1 and Lemma 1)
We have $(Traces_{Vul}(\mathcal{V}) \cap (Traces_{L_{S1^!_{comp}}}(S1^!_{comp}) \cup Traces_{L_{S2^!_{comp}}}(S2^!_{comp}))) \cap Traces(CUT^\delta) \neq \emptyset$. Hence, $Traces_{Vul}(\mathcal{V}) \cap Traces(CUT^\delta) \neq \emptyset$ (a) and $(Traces_{L_{S1^!_{comp}}}(S1^!_{comp}) \cup Traces_{L_{S2^!_{comp}}}(S2^!_{comp})) \cap Traces(CUT^\delta) \neq \emptyset$ (b). From (a), we have $CUT \not\models \mathcal{V}$ (Definition 4). Consequently, $CUT$ is vulnerable to $\mathcal{V}$.
If $\sigma \in Traces_{VUL/FAIL}(tc)$ then, from (b) we have $(Traces_{Fail}(S1^!_{comp}) \cup Traces_{Fail}(S2^!_{comp})) \cap Traces(CUT^\delta) \neq \emptyset$. We obtain $CUT \not\models \mathcal{S}_{comp}$ (Definition 8).

The proofs of 3,4,5 are similar.

■

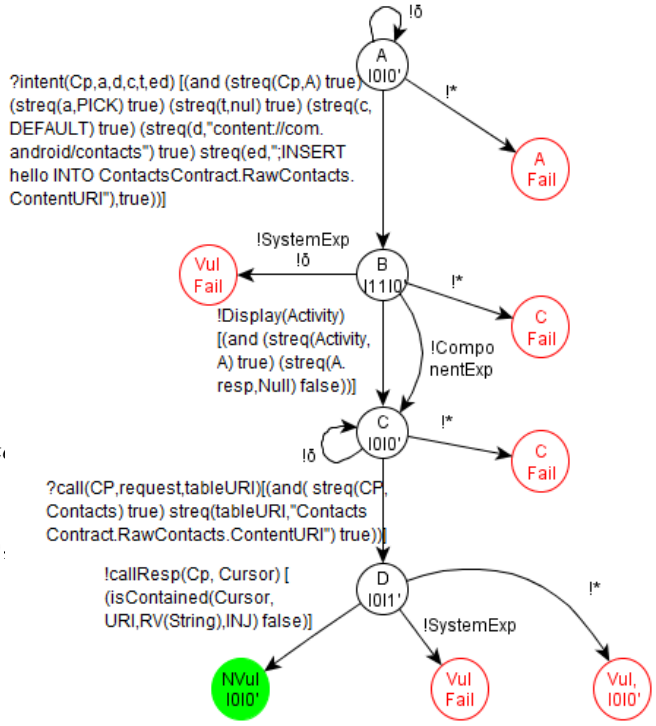


**Fig. 10.** A test case example

*Example 4.* A test case example, derived from the specification of Figure 9 and the vulnerability pattern of Figure 6 is depicted in Figure 10. Firstly, it stems from the parallel product of the specification and of the vulnerability pattern which keeps the shared behaviours of the two initial ioSTS and also composes their locations together. Then, this product is concretised with Algorithm 3 (with the String values carried out by the test case). This test case reflects the sending of an implicit intent to an Activity with the extra data parameter composed of an SQL injection. After receiving this intent, the Activity behaves as expected if it produces an exception or returns a screen with a response. Otherwise, the test execution is terminated: if the Activity crashes (modelled by !SystemExp) or if quiescence is observed, the location $(Vul, Fail)$, which belongs to the verdict location set VUL/FAIL, is reached. This means the Activity is vulnerable and does not respect the Android documentation (verdict VUL/FAIL). The verdict is FAIL when observing any other action (unexpected action, response=null) since the location $(C, Fail) \in FAIL$ is reached. The test execution is stopped because the component is not functioning correctly. Afterwards, the test case checks if the data managed by the ContentProvider Contacts have not been updated after the sending of the intent. This is checked with the transitions carrying the action ?call and !callResp. If the ContentProvider does not return values which belong to the sets used for testing ($URI$, $RV$,$INJ$) then it is not vulnerable (location $(NVUL, l0l0) \in NVUL$ reached and test verdict

```
  public void test1() {
2         assertFalse(FAIL,isDisplay());
          assertFalse(FAIL, hasResponseFrom
4             ContentProvider());

6         // Intent
          mIntent.setAction("android.intent.action.
              PICK");
8         mIntent.addCategory("android.intent.
              category.DEFAULT");
          mIntent.setData(Uri.parse("content://com.
              android.contacts"));
10        mIntent.putExtra("contact", ";INSERT
              hello INTO ContactsContract.
              RawContacts.ContentURI");
          setActivityIntent(mIntent);
12        try {
              mActivity = getActivity();
14            assertNotNull(VUL/FAIL, mActivity);
              assertTrue(VUL/FAIL, isDisplay());
16            // PICK : expect response
              ActivityResult response = monitor.
                  getResult();
18            assertTrue(FAIL,solve("(and (
                  isDisplay() true) (streq(
                  response.getResultData(),null)
                  false))"));
              getInstrumentation().
                  callActivityOnStop(mActivity);
20        } catch (Exception e) {
              assertTrue(true);
22        }

24        assertFalse(FAIL,isDisplay());
          assertFalse(FAIL,
              hasResponseFromContentProvider());
26
          /**
28         * call the ContentProvider
           */
30
          try {
32          callCp(Uri.parse("content://
                ContactsContract.RawContacts.
                ContentURI"));
            assertTrue(VUL,
                hasResponseFromContentProvider());
34          assertTrue(VUL, solve("(and (
                isContained(cursor, URISet) false)
                 (isContained(cursor, RVSet) false
                ) (isContained(cursor, INJSet)
                false))"));
            cursor = null;
36             } catch (Exception e2) {
              fail(VUL);
38        }
          assertFalse(VUL, isDisplay());
40    }
```

**Fig. 11.** A JUNIT test case

NVUL). If it crashes it is vulnerable and does not meet its specification as well (verdict VUL/FAIL). Otherwise, it is vulnerable. A FAIL verdict is also given when an abnormal action is received before calling the Activity with an intent or just before calling the ContentProvider. These extra actions should not be received and might perturb the testing process. For instance, an Activity cannot return an action without being called with any intent before. If this really happens then the testing process is abnormal and is terminated to avoid false positives.

ioSTS test cases are then converted into JUNIT test cases in order to be executed with our framework. This conversion can be summarised by the following steps:

- transitions labelled with input actions are converted into component calls. For example, a transition $l \xrightarrow{?intent(Cp,a,d,c,t,ed),G,A} l2$ is converted into the sending of an intent composed of parameter values given in the guard $G$,
- the transitions carrying output actions are translated into Java code and JUNIT assertions. For an output action $!a(p)$ which does not correspond to the raise of an exception, we assume having a corresponding Boolean function a() which is called inside an assertion. The verdicts *VUL/FAIL*, *VUL* and *FAIL* are determined from assertions of the form *Assert(mess, expr)* returning the message *mess* equal to the verdict when *expr* is false. The *NVUL/FAIL* verdict is obtained from the failed assertions that are not composed of message. By deduction, the *NVUL* verdict is set if there is no failed assertion during the testing process.
- a transitions $l \xrightarrow{!ComponentExp} l_f$, is converted into a *try/catch* statement composed of an assertion producing the verdict carried by $l_f$. A transition $l \xrightarrow{!SystemExp} lf$, modelling a component crash is not converted into JUNIT code but is replaced by try/-catch statements in the test runner call. If the test runner catches an exception, the latter is converted into an assertion composed of the messages *VUL* or *VUL/FAIL* according to the label of the location $lf$.

*Example 5.* As an example, Figure 11 gives the JUNIT test case derived from the ioSTS of Figure 10. For readability, we removed the initial declarations and instantiations. Initially, the incorrect receipt of output actions is coded with the two assertions of lines (2-4) (and later with lines (24-26)). The intent action is translated into the call of the component mActivity (lines 7-11). Lines (12-22) represent the assertions derived from the different output actions that may be received after the intent. The transitions starting from $(Bl11l0')$ in the test case of Figure 10 are translated by the assertions in lines (14-18). For instance, the assertion in line 15 fails with the VUL/FAIL message if Display() returns false. The next assertion in line 18 fails with the FAIL message if a screen is displayed and if a null response is provided by the activity. This more complex guard is computed with the *solve* function. The transition $B(l11l0') \xrightarrow{!ComponentExp} (Cl0l0')$ of the ioSTS test case is converted into the catch block composed of an assertion always true (line 21). The ContentProvider Contacts is then called to extract all the data of the table *RawContacts* (line 32). If these data are not composed of values which belong to $INJ \cup RV \cup URI$ (lines 33-34) and if no other action (exception or screen) is observed (lines 38-41) then no assertion fails. The verdict is NVUL. Otherwise, at

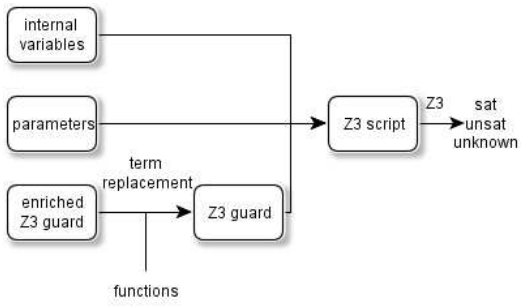


**Fig. 12.** The satisfiability function

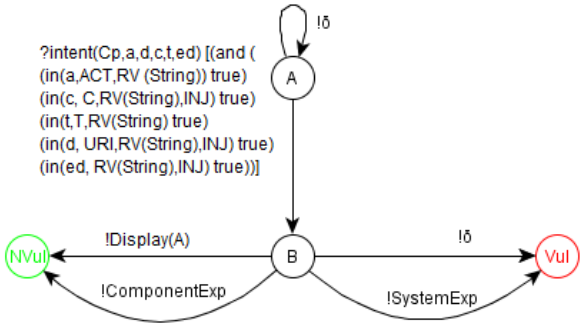


**Fig. 13.** Vulnerability pattern $\mathcal{V}1$

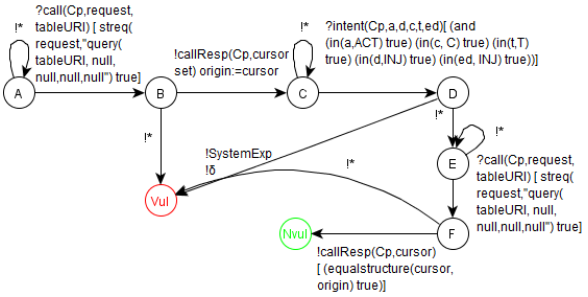


**Fig. 14.** Vulnerability pattern $\mathcal{V}3$

least one assertion of lines( 31-41) fails and the verdict becomes VUL.

Once the test case execution is completed, an XML report is written by the Test runner which lists all the failed assertions. The report, we have depicted in Figure 5, were obtained after the execution of the previous JUNIT test case. After the intent call, the component crashed with the NullPointerException exception. An assertion failed with the message VUL/FAIL. Consequently, the test verdict is VUL/FAIL.

The guard solving, used in Algorithm 3 and during the test case execution, is performed by the Z3 solver whose language is augmented with the predicates given in Section 3. Below, we give more details about the guard solving.

### *4.4 Satisfiability of guards*

We have chosen the SMT (Satisfiability Modulo Theories) solver Z3 [23] for checking the satisfiability of guards since it offers good performance, takes several variable types (Boolean, Integer, etc.) and allows a direct use of arithmetic formulae. However, it does not accept String variables whereas the String type is widely used with Android. We have chosen to extend the SMT-LIB language used by Z3 with new predicates instead of using a String constraint solver like Hampi [17]. After some experimentations, we indeed concluded that our extension of Z3 offers better performance than using a combination of solvers or even Hampi alone.

These new predicates stand for functions over internal variables and parameters which must return a Boolean value. We implemented several predicates to consider String variables e.g., *streq(String,String)* or *contains (String,String)* to check the equality of two strings or if the second string passed in is contained in the first one. ***Users can update the current predicate list by adding, for each new predicate, a method returning a Boolean to the class ??? into the APSET source code.***

The call of Z3 is performed with a satisfiability function which takes an internal variable valuation set, the parameter valuations received with messages and a guard written with an enriched SMT-LIB expression. Its functioning is summarised in Figure 12. Predicates referring to Boolean functions are computed and replaced with Boolean values. A Z3 script, composed of the internal variable valuations, the parameter valuations received with an action, is then dynamically written by the satisfiability function before calling Z3. If the guard is satisfiable (not satisfiable), Z3 returns *sat* (*unsat* respectively). Z3 returns *unknown* when the satisfiability of a formula is not decidable.

For instance, the guard $(resp \neq Null) \wedge (resp =$ "*done*") is written with the expression *(and (strneq(Resp, Null) true) (streq(Resp,"done") true ))*. The predicates *strneq* and *streq* are computed with the values of the variables resp and replaced by Boolean variables. Then, a Z3 script is written and executed.

## 5 Experimentation

The experimentation was performed with the tool APSET on 70 Android applications: 20 applications still under development provided by the Openium company [4], and 50 popular applications randomly chosen in the Android Market. Test cases were generated from four vulnerability patterns. The first one $\mathcal{V}1$ targets the availability of Activities. The others aim at checking confidentiality and integrity:

[4] http://www.openium.fr/

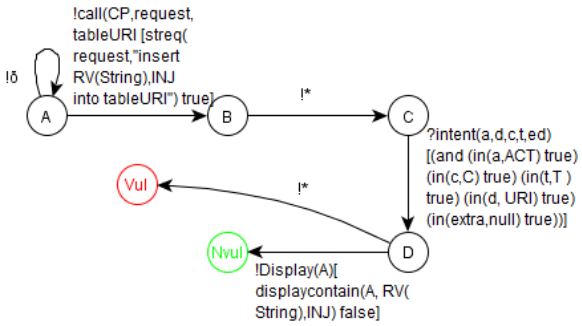


**Fig. 15.** Vulnerability pattern $\mathcal{V}4$

- $\mathcal{V}1$, illustrated in Figure 13, expresses that an Activity is vulnerable when it becomes unavailable after the sending of intents composed of String values known for relieving bugs (*RV* set) or of SQL injections. With this vulnerability pattern, an Activity is vulnerable when quiescence is observed or when the Activity crashes (action !SystemExp),
- the second vulnerability pattern $\mathcal{V}2$ is the one taken as example in Figure 6,
- $\mathcal{V}3$, depicted in Figure 14, is an extension of the previous vulnerability pattern that aims at checking whether the structure of a database managed by a ContentProvider is not altered (modification of attribute names, removal of tables, etc.) by SQL injections forwarded by an Activity receiving intents. The main difference between $\mathcal{V}2$ and $\mathcal{V}3$ concerns the call of the ContentProvider with the three first transitions to extract the database structure that is stored in the variable origin. After the sending of an intent composed of SQL injections, the component is not vulnerable if the database structure is equal to origin (use of a new predicate *equalstructure*),
- $\mathcal{V}4$, illustrated in Figure 15, aims at checking that incorrect data, already stored in a database, are not displayed by an Activity after having called it with an intent. Incorrect data (in the set $RV(String) \cup INJ$) are initially stored into the database (two first transitions). Whatever the response provided by the ContentProvider, the Activity is not vulnerable if the content of its screen is not composed of the incorrect data previously stored. The comparison of the content of the screen with the set $RV(String) \cup INJ$ is performed with the predicate *displaycontain*.

With these vulnerability patterns, APSET detected a total of 62 vulnerable applications (88 %) out of the 70 tested applications. Among the 50 tested applications available on the Android market, 41 have security defects which are more or less serious (from application crash up to personal data extraction). APSET generated a average of 303 test cases per application and detected an average of 70 *VUL* verdicts. Figure 16 illustrates the test results of 30 randomly chosen applications among the 50 of the Android Market. This histogram shows

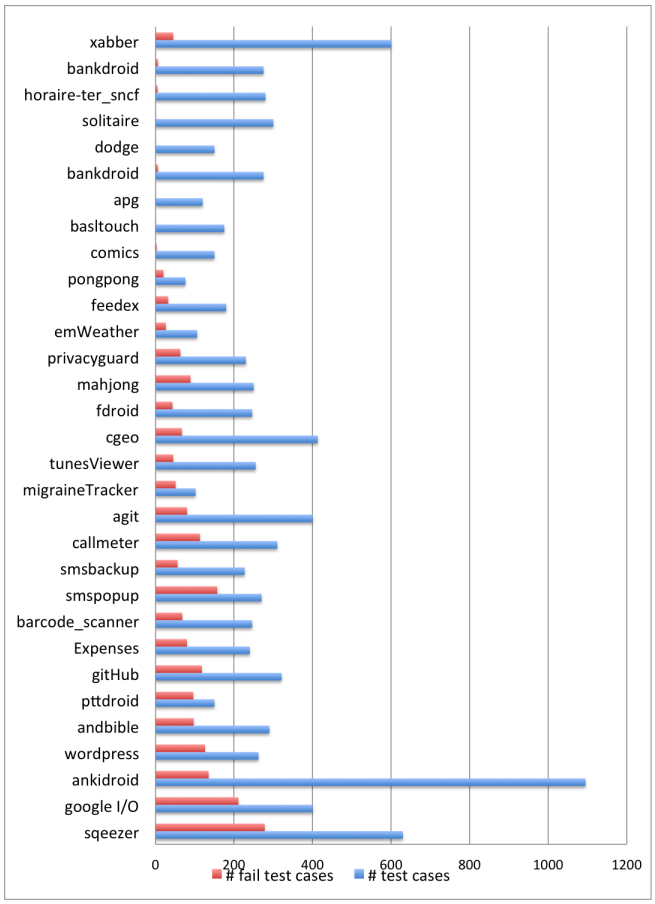


**Fig. 16.** Test results on 30 applications

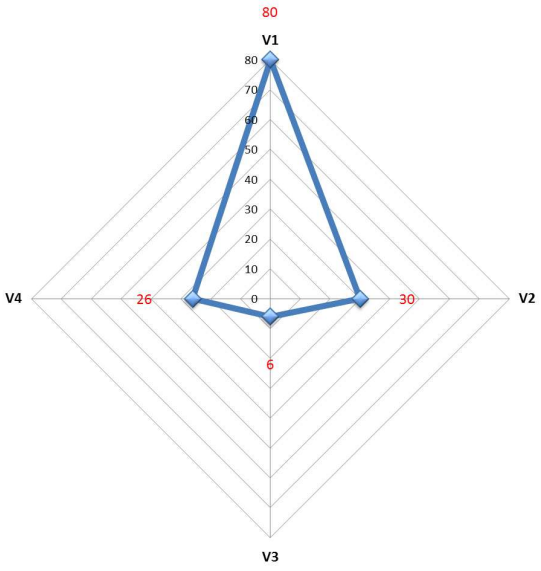


**Fig. 17.** Percentage of vulnerable applications / vulnerability pattern

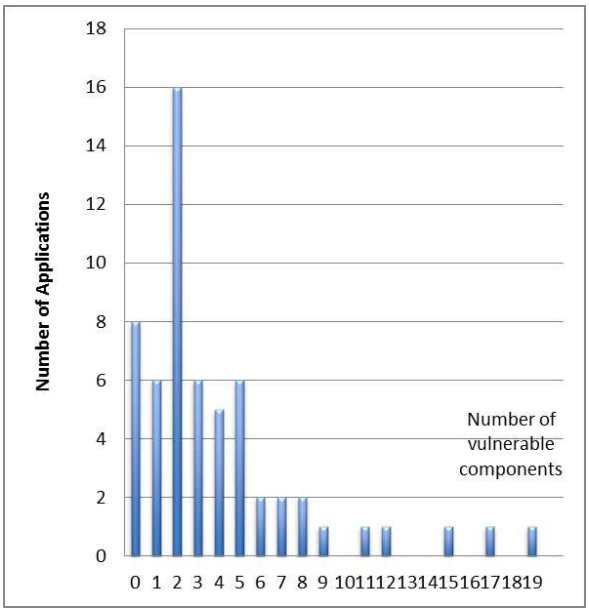


**Fig. 18.** Number of vulnerable components per application

| Applications | | $\mathcal{V}1$ test results | |
|---|---|---|---|
| Name | # component | *#vul/ #testcases* | Time/ test |
| app 1 | 35 | 861/969 | 8s |
| app 2 | 6 | 95/147 | 12s |
| app 3 | 5 | 0/117 | 4s |
| app 4 | 24 | 52/545 | 0.15s |
| app 5 | 11 | 3/33 | 2s |
| app 6 | 11 | 11/120 | 3s |
| app 7 | 11 | 20/110 | 3s |
| app 8 | 11 | 20/110 | 3s |
| app 9 | 13 | 19/80 | 0.90s |
| app 10 | 16 | 15/105 | 2.1s |
| app 11 | 9 | 15/213 | 3.11s |
| app 12 | 7 | 66/311 | 2.7s |
| app 13 | 8 | 0/300 | 0.27s |
| NotePad | 5 | 44/300 | 0.12s |
| SearchableDictionnary | 3 | 34/288 | 0.97s |
| Google Maps | 38 | 31/300 | 1.92s |
| Youtube | 12 | 19/336 | 4.1s |

**Table 2.** Experimentation Results with $\mathcal{V}1$

| Applications | | Test results | | | | |
|---|---|---|---|---|---|---|
| Name | #component | $\mathcal{V}2$ | $\mathcal{V}3$ | $\mathcal{V}4$ | #vul/# test-cases | Time/ test |
| app2 | 7 | 8 | 0 | 3 | 11/54 | 0,85 |
| app5 | 15 | 10 | 2 | 5 | 17/102 | 2,3 |
| app11 | 16 | 1 | 0 | 3 | 4/164 | 0,18 |
| app12 | 9 | 2 | 0 | 9 | 11/97 | 1,56 |
| app13 | 7 | 19 | 0 | 2 | 21/73 | 0,77 |
| app14 | 8 | 0 | 0 | 4 | 4/71 | 1,05 |
| NotePad | 5 | 27 | 0 | 4 | 31/44 | 0,04 |
| SearchableDictionnary | 3 | 10 | 0 | 3 | 13/22 | 1,02 |
| Google Maps | 38 | 28 | 0 | 11 | 39/370 | 1,67 |
| Youtube | 12 | 3 | 0 | 0 | 3/131 | 3,21 |

**Table 3.** Experimentation results with $\mathcal{V}2$,$\mathcal{V}3$,$\mathcal{V}4$

the number of test cases executed per application and the number of *VUL* verdicts obtained. Some application test results revealed a high number of *VUL* verdicts. The latter do not necessarily reflect the number of security defects though. Several *VUL* verdicts arised on account of the same defect in the source code. The analysing of XML reports should help developers localise these defects by identifying the incriminated components, the raised exceptions or the actions performed. Most of the failed applications have a number of faulty components varying between 1 and 5. The distribution of the number of vulnerable components per application is depicted in Figure 18. The radar chart of Figure 17 also depicts the percentage of applications vulnerable to the vulnerability patterns $\mathcal{V}1, ..., \mathcal{V}4$. Among the 62 vulnerable applications, 80 % have availability issues detected with $\mathcal{V}1$. In other words, they are no longer available when receiving intents composed of incorrect values or SQL injections. 36 % are vulnerable to SQL injections ($\mathcal{V}2$ and $\mathcal{V}3$). 26 % of the applications are composed of Activities displaying incorrect data without validation ($\mathcal{V}4$).

We manually analysed the test reports of 18 applications for $\mathcal{V}1$ and 10 applications for $\mathcal{V}2, ..., \mathcal{V}4$ and checked these results with the source codes. The applications app1 to app14 are provided by the Openium company.

Table 2 summarises the results observed with $\mathcal{V}1$. For each application, Table 2 lists the number of tested components, the number of vulnerability issues detected over the test case number and the average test case execution time delay. For instance, 969 test cases were generated by our tool for *app 1* and 861 revealed issues. All these issues were essentially observed by Activity crashes when receiving malicious intents (receipt of exceptions such as *NullPointerException*).

Table 3 depicts the results obtained on 10 applications extracted from Table 2 after having experimented them with the vulnerability patterns $\mathcal{V}2$, $\mathcal{V}3$ and $\mathcal{V}4$. Table 3 shows respectively the number of tested components, the number of issues detected with each vulnerability pattern and the total number of test cases providing a vulnerable verdict. Table 3 does not list all the applications of Table 2 since $\mathcal{V}2$, $\mathcal{V}3$ and $\mathcal{V}4$ can only be experimented on applications composed of ContentProviders. For instance, with *app5*, 102 test cases

were generated and 17 showed vulnerability issues. 10 test cases showed that *app5* is vulnerable to $\mathcal{V}2$. More precisely, we noticed that 1 test case showed that personal data can be modified by using malicious intents. *app5* crashed with the other test cases, probably because of the bad handling of malicious intents by the components. 2 test cases also revealed that the structure of the database can be modified ($\mathcal{V}3$). By analysing the ContentProvider source codes of *app5*, we actually detected that no ContentProvider methods were protected against malicious SQL requests. Finally, 5 test cases revealed the display of incorrect data stored in database ($\mathcal{V}4$). This means that the database content is straight displayed into the user interface.

With the application Google Maps, 39 test cases revealed vulnerability issues. 28 vulnerabilities were detected with $\mathcal{V}2$: 3 (resp. 1) test cases showed that databases can be updated or modified with incorrect data through the component *MapsActivity* (resp. *ResolverActivity*). Personal data can be exposed and modified with SQL injections, therefore Google Maps has integrity issues. The other failed test cases concern the crashes of *MapsActivity*, with the exceptions NullPointerException or RuntimeException. For the vulnerability pattern $\mathcal{V}4$, the 11 detected defects also correspond to component crashes. For Youtube, we obtained 3 VUL/FAIL verdicts resulting from component crashes with the exception NullPointerException. No more serious vulnerability were detected here.

Tables 2 and 3 also gives the average test case execution time, measured with Mid 2011 computer with a CPU 2.1Ghz Core i5 and 4GB of RAM. Each test case execution took few seconds for most of them. Some required longer time processing than others though (some milliseconds up to 12s). This difference comes from the application code. For instance, for *app 1*, some Activities perform several successive tasks: the receipt of an intent, the call of a ContentProvider to insert data into database, the call of a remote Web Service via a Service component and finally the display of a screen. Testing these Activities requires a longer execution time than testing other components such as the Activities of *app 4* which simply displays a screen. Nonetheless, the longer test case execution times do not exceed few seconds. These results are coherent with other available Android application testing tools. For instance, Benli et al. [4] showed in their studies that the execution time per test case may be up to 17s. These results combined with the number of vulnerability issues detected on real applications tend to show that our tool is effective and leads to substantial improvement in security vulnerability detection.

## 6 Related work

Security testing and Android security improvements are not new trends. Below, we compare our approach with some recent works from the literature.

Originally, security testing, based upon formal models, has been studied in several works. For instance, Letraon et al. proposed a test generation technique to check whether security rules modelled with the OrBAC language hold on implementations under test [18]. A mutation testing technique is considered for testing the access control policy robustness. The method proposed in [21] generates test cases from a specification and invariants or rules describing security policies. These two works assume having a specification or a test case set as inputs. In contrast, these assumptions are not required in our proposal. Instead, we propose a partial specification generation for Android components. Mouelhi et al. introduced a test case generation for Java applications from security policies described with logic-based languages e.g., OrBAC, to describe access control properties [22]. Policy enforcement points are injected into the code which is later tested with a mutation testing approach. Our work is not dedicated to access control policy. We also do not modify the original code. Furthermore, instead of considering a mutation technique to concretise test cases, we combine the use of SQL, XML injections, values known for relieving bugs and random testing. The use of these sets should increase the defect detection rate.

Marback et al. proposed a threat modelling based on trees in [20]. This method produces test cases from threat trees and transforms them into executable tests. Although the use of trees is intuitive for Industry, formal models offer several other advantages such as the description of the testing verdict without ambiguity. Furthermore, specifications are not considered in this work, so false positive or negative results may be discovered with a higher rate than with our method.

Other works, dealing with Android security, have been also proposed recently. Some works focused on the definition of a more secure Android system. The approaches proposed in [24,27] checks system integrity with monitoring techniques. The tool ProfileDroid monitors several layers (user interfaces, networks, etc.) and produces metrics on network traffic, user events, OS system calls, etc. This kind of tool helps uncover undesired behaviour and some vulnerabilities e.g., unencrypted networks, that are not considered in this paper.

More recently, several efforts have been made to investigate privilege problems in Android applications [10, 11,5]. For instance, the tool Stowaway was also developed to detect over privilege in Android applications [11]. It statically analyses application codes and compares the maximum set of permissions needed for an application with the set of permissions actually requested. This approach offers a different point of view, in comparison to our work, since we focus on applications taking

place over the system. We assume that the right permissions are granted to components and that there is no overprivilege issue. Amalfitano et al. proposed a GUI crawling-based testing technique of Android applications [3] which is a kind of random testing technique based on a crawler simulating real user events on the user interface and automatically infers a GUI model. The source code is instrumented to detect defects. This tool can be applied on small size applications to detect crashes only. In contrast, APSET detects a larger set of vulnerabilities since it takes vulnerability scenarios on Activities, but also on Services and ContentProviders. However, APSET does not directly handle sequences of Activities.

The analysis of the Android intent mechanism were also studied in [7]. Chin et al. described the permission system vulnerabilities that applications may exploit to perform unauthorised actions. Vulnerability patterns, that can be used with our method, can be extracted from this work. The same authors also proposed the tool Comdroid which analyses Manifest files and application source codes to detect weak permissions and potential intent-based vulnerabilities. Nevertheless, the tool provides a high rate of false negatives (about 75 %) since it only warns users on potential issues but does not verify the existence of security flaws. The tool Fuse, proposed in [14], roughly offers the same features as Comdroid and the same disadvantages. APSET actually completes Comdroid and Fuse since it tests vulnerability issues on blackbox applications. Another way to reduce intent-based vulnerabilities is to modify the Android platform. In this context, Kantola et al. proposed to upgrade the heuristics that Android uses to determine the eligible senders and recipients of messages [16]. Explicit intents are passed through filters to detect those unintentionally sent to third-party applications. With a modified version of Comdroid, the authors show that applications are less vulnerable to attacks. It is manifest that all attacks are not blocked by these filters, so security testing is still required here. Furthermore, ContentProviders and the management of personal data is not considered. Other studies deal with the security of pre-installed Android Applications and show that target applications receiving oriented intents can re-delegate wrong permissions [29,13]. Some tools have been developed to detect the receipt of wrong permissions by means of malicious intents. In our work, we consider vulnerability patterns to model more general threats based on availability, integrity or authorisation, etc. Permissions can be taken into consideration in APSET with the appropriate vulnerability patterns. The robustness of inter-component communication in Android was studied in [19]. In short, components are tested with intents composed of values known for relieving bugs. Components are not robust when observing crashes and bad exception handling. Our approach includes this kind of robustness technique with the vulnerability pattern $\mathcal{V}1$ described in the experimentation. Jing et al. introduced a model-based conformance testing framework for the Android platform [15]. Like in our approach, partial specifications are constructed from Manifest files. Nevertheless, the authors do not consider the Android documentation to augment the expressiveness of these specifications and consider implicit intents only. Test cases are generated, from these specifications, to check whether intent-based properties hold. This approach lacks of scalability though since the set of properties, provided in the paper, is based on the intent functioning and cannot be modified. Our work takes as input a larger set of vulnerability patterns.

Finally, in [28], we presented a rudimentary introduction of this work by presenting the vulnerabilities exposed by the Android intent mechanism. We also presented the insight of this testing methodology but without describing the specification generation, the test case generation and execution.

## 7 Conclusion

In this paper, we presented APSET, an Android aPplications SEcurity Testing tool for detecting intent-based vulnerabilities on Android components. This tool takes vulnerability patterns proposed by an Android expert. It automatically generates and then executes test cases on smartphones or emulators. The main contribution of this work resides in the test case generation by means of an automatic generation of partial specifications from Android applications. These are used to generate test cases composed of either implicit or explicit intents. These specifications also contribute to complete the test verdict with the specific verdicts *NVUL/FAIL* and *VUL/-FAIL*, pointing out that the component under test does not meet the recommendations provided in the Android documentation. They also reduce false positive verdicts since each component is exclusively tested by means of the vulnerability patterns that share behaviour with the component specification. Test cases, derived from vulnerability patterns, that cannot be theoretically experimented on a component, are not generated nor executed. APSET can also detect data vulnerabilities based on the intent mechanism since it supports the testing of compositions of Activities or Services with ContentProviders. We experimented APSET on 70 Android applications. With only 4 vulnerability patterns, we detected that 62 applications have defects that can be exploited by attackers to crash applications, to extract personal data or to modify them.

This experimentation firstly showed that APSET is effective and can be used in practice. The APSET effectiveness could be improved by using more vulnerability patterns or eventually by generating more test cases per pattern. Indeed, to limit the test cost, the test case number is bounded in APSET. This number can be modified in the tool.

Secondly, compared to the available intent-based testing tools [7,15,16], APSET is scalable since existing vulnerability patterns can be modified or new vulnerability patterns can be proposed to meet the testing requirements. Value sets used for testing, e.g., the SQL injection set *INJ* and the set of predicates used in guards can be updated as well.

Lastly, APSET is simple to use. Vulnerability patterns are stored in DOT files. Various tools can process DOT files and hence graphically represent vulnerability patterns. Then, test cases are automatically generated and executed. The intermediate models e.g., the partial specifications, are stored in DOT files and can be viewed as well. XML reports, that can be used with continuous integration tools, also list the issues encountered while testing and give the number of vulnerabilities detected.

Nevertheless, APSET is only based on the intent mechanism. So, any kind of vulnerability cannot be tested with APSET. For instance, attack scenarios composed of several actions performed on the application interface, cannot be applied with our tool. The latter does not yet consider the component type BroadcastReceiver. This component type is also vulnerable to malicious intents though. These drawbacks could be explored in future works.

Another direction of future research concerns the generation of partial specifications from implementations (classes, configuration files, etc.) or documentations or expert knowledges. As stated previously, these partial specifications can participate in the test case generation to refine the test verdict and to reduce false positives. They could also be exploited for generating documentation. In this work, partial specifications, composed of some actions, are built with algorithms derived from the recommendations provided in the Android documentation. If the partial specification construction algorithms are upgraded with a lot of actions, the risk to produce false specifications and hence false test cases could drastically increase. We simply solved this problem here by considering straightforward algorithms that were evaluated by Android experts. A longer study could raise new and more formal methods of partial specification generation which could even take into consideration the risk to produce incorrect models.

## References


1. Android developer page. In: http://developer.android.com/index.html, last accessed feb 2013
2. It business: Android security. In: http://www.itbusinessedge.com/cm/blogs/weinschenk/google-must-deal-with-android-security-problems-quickly/?cs=49291, last accessed feb 2013 (2012)
3. Amalfitano, D., Fasolino, A., Tramontana, P.: A gui crawling-based technique for android mobile application testing. In: In IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 252–261. IEEE (2011)
4. Benli, S., Habash, A., Herrmann, A., Loftis, T., Simmonds, D.: A comparative evaluation of unit testing techniques on a mobile platform. In: Proceedings of the 2012 Ninth International Conference on Information Technology - New Generations, ITNG '12, pp. 263–268. IEEE Computer Society, Washington, DC, USA (2012)
5. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: 19th Annual Network & Distributed System Security Symposium (NDSS) (2012)
6. Chaudhuri, A.: Language-based security on Android. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pp. 1–7. ACM, New York, NY, USA (2009)
7. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proc. of the 9th International Conference on Mobile Systems, Applications, and Services (2011)
8. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proc. of the 25th International Conference on Software Engineering, pp. 38–48 (2003)
9. Cuppens, F., Cuppens-Boulahia, N., Sans, T.: Nomad : A security model with non atomic actions and deadlines. In: Computer Security Foundations. CSFW-18 2005. 18th IEEE Workshop, pp. 186–196 (2005)
10. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Proceedings of the 13th international conference on Information security, ISC'10, pp. 346–360. Springer-Verlag (2011)
11. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: In Proceedings of the 18th ACM conference on Computer and communications security, pp. 627–638 (2011)
12. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In: J. Grabowski, B. Nielsen (eds.) FATES 2004, no. 3395 in Lecture Notes in Computer Science, pp. 1–15. Springer (2005)
13. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock Android smartphones. In: Proceedings of the 19th Network and Distributed System Security Symposium (NDSS) (2012)
14. Hurd, J.: "fuse: Inter-application security for android". In: Proceedings of the High Confidence Software & Systems (HCSS 2012) (2012)
15. Jing, Y., Ahn, G.J., Hu, H.: Model-based conformance testing for android. In: G. Hanaoka, T. Yamauchi (eds.) Proceedings of the 7th International Workshop on Security (IWSEC), *Lecture Notes in Computer Science*, vol. 7631, pp. 1–18. Springer (2012)
16. Kantola, D., Chin, E., He, W., Wagner, D.: Reducing attack surfaces for intra-application communication in android. In: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, SPSM '12, pp. 69–80. ACM, New York, NY, USA (2012)
17. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis, pp. 105–116. ACM, New York, NY, USA (2009)

18. Le Traon, Y., Mouelhi, T., Baudry, B.: Testing security policies: going beyond functional testing. In: ISSRE'07 (Int. Symposium on Software Reliability Engineering) (2007)
19. Maji, A., Arshad, F., Bagchi, S., Rellermeyer, J.: An empirical study of the robustness of inter-component communication in android. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on, pp. 1–12 (2012)
20. Marback, A., Do, H., He, K., Kondamarri, S., Xu, D.: A threat model-based approach to security testing. Software: Practice and Experience **43**(2), 241–258 (2013)
21. Marchand, H., Dubreil, J., Jéron, T.: Automatic Testing of Access Control for Security Properties. In: TESTCOM/FATES 2009 (2009)
22. Mouelhi, T., Fleurey, F., Baudry, B., Traon, Y.: A model-based framework for security policy specification, deployment and testing. In: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, pp. 537–552 (2008)
23. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: C.R. Ramakrishnan, J. Rehof (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
24. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in android. In: Proceedings of the 2009 Computer Security Applications Conference, ACSAC '09, pp. 340–349 (2009)
25. OWASP: Owasp testing guide v3.0 project (2003). URL `http://www.owasp.org/index.php/Category:OWASP\_Testing\_Project\#OWASP\_\\Testing\_Guide\_v3`
26. Sen, K.: Generating optimal monitors for extended regular expressions. In: Proc. of the 3rd Workshop on Runtime Verification (RV03), volume 89 of ENTCS, pp. 162–181. Elsevier Science (2003)
27. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Profiledroid: multi-layer profiling of android applications. In: Proceedings of the 18th annual international conference on Mobile computing and networking, Mobicom '12, pp. 137–148. ACM, New York, NY, USA (2012)
28. Zafimiharisoa, S.R., Salva, S., Laurençot, P.: An automatic security testing approach of android applications. In: The Seventh International Conference on Software Engineering Advances (ICSEA 2012), pp. 643–646. XPS (Xpert Publishing Services), Lisbon, Portugal (2012)
29. Zhong, J., Huang, J., Liang, B.: Android permission redelegation detection and test case generation. In: Computer Science Service System (CSSS), 2012 International Conference on, pp. 871 –874 (2012)